# 기술동향분석보고서

2003 년 2 사분기

## Autonomic Computing

2003 년 8 월 13 일

하 병 언

한 국 I B M

# Contents

B. Autonomic Computing 2: Implications for IT services
C. Autonomic computing and IBM
D. Autonomic computing and the Tivoli software storage family
E. Autonomic personal computing
F. Clockwork: A new movement in autonomic systems
G. Comparing autonomic and proactive computing
H. Dynamic reconfiguration: Basic building blocks for autonomic computing on IBM pSeries servers
I. How Tivoli software products support the IBM Autonomic Computing Initiative$^{tm}$
J. Autonomic computing: Can it help to manage the increasingly complex information environment?
K. IBM Global Services and autonomic computing
L. The dawning of the autonomic computing era
M. The Vision of Autonomic Computing
N. The Tivoli software implementation of autonomic computing guidelines

# Preface

I/T developed brilliantly for last several decades, but accordingly, a lot of problems occurred. The millions of businesses, billions of humans that compose them, and trillions of devices that they will depend upon all require the services of the I/T industry to keep them running. And it's not just a matter of numbers. It's the complexity of these systems and the way they work together that is creating a shortage of skilled I/T workers to manage all of the systems. It's a problem that's not going away, but will grow exponentially, just as our dependence on technology has.

The solution may lie in automation, or creating a new capacity where important computing operations can run without the need for human intervention. On October 15th, 2001 Paul Horn, senior vice president of IBM Research addressed the Agenda conference, an annual meeting of the preeminent technological minds, held in Arizona. In his speech, and in a document he distributed there, he suggested a solution: build computer systems that regulate themselves much in the same way our autonomic nervous system regulates and protects our bodies.

This new model of computing is called autonomic computing. The good news is that some components of this technology are already up and running. However, complete autonomic systems do not yet exist. This is not a proprietary solution. It's a radical change in the way businesses, academia, and even the government design, develop, manage and maintain computer systems. Autonomic computing calls for a whole new area of study and a whole new way of conducting business. This document is that investigate the recent autonomous computing technology trend.

# Autonomic computing concepts

e-business

## Introducing autonomic computing

Autonomic computing systems have the ability to manage themselves and dynamically adapt to change in accordance with business policies and objectives. Self-managing systems can perform management activities based on situations they observe or sense in the IT environment. Rather than IT professionals initiating management activities, the system observes something about itself and acts accordingly. This allows the IT professional to focus on high-value tasks while the technology manages the more mundane operations.

## Why autonomic computing?

Why is autonomic computing important today? The cost of technology continues to decrease yet overall IT costs do not. With the expense challenges that many companies face, IT managers are looking for ways to improve the return on investment of IT by reducing total cost of ownership, improving quality of service, accelerating time to value and managing IT complexity. e-business is a reality, and outages are proving to be expensive and embarrassing.

Adding to this complexity is the proliferation of heterogeneous vendor and technology environments, which require the components of a given solution to be integrated and customized into unique customer business processes. The increased need to distribute data, applications and system resources across geographic and business boundaries further contributes to the complexity of the IT infrastructure. The additional complexity keeps the costs of managing (deploying, tuning, fixing, securing) the IT infrastructure high.


**Introduction of self-management**

Systems with self-managing components reduce the cost of owning and operating computer systems. IT infrastructure components take on the following characteristics: self-configuring, self-healing, self-optimizing and self-protecting.

***Self-configuring***

With the ability to dynamically configure itself on the fly, an IT environment can adapt immediately—and with minimal human intervention—to the deployment of new components or changes in the IT environment. Dynamic adaptation helps verify continuous strength and productivity of an e-business infrastructure—often the single determining factor between business growth and chaos.

***Self-healing***

Self-healing IT environments can detect improper operations (either proactively through predictions or otherwise) and then initiate corrective action without disrupting system applications. Corrective action could mean that a product alters its own state or influences changes in other elements of the environment. Day-to-day operations do not falter or fail because of events at the component level. The IT environment as a whole becomes more resilient as changes are made to reduce or help eliminate the business impact of failing components.

***Self-optimizing***

Self-optimization refers to the ability of the IT environment to efficiently maximize resource allocation and utilization to meet end users' needs with minimal human intervention. In the near term self-optimization primarily addresses the complexity of managing system performance. In the long term self-optimizing components may learn from experience and automatically and proactively tune themselves in the context of an overall business objective. Self-optimization verifies optimum Quality of Service for both system users and their customers.

***Self-protecting***

The goal of self-protecting environments is to provide the right information to the right users at the right time through actions that grant access based on the users' role and pre-established policies. A self-protecting IT environment can detect hostile or intrusive behavior as it occurs and take autonomous actions to make itself less vulnerable to unauthorized access and use, viruses, denial-of-service attacks and general failures. The self-protecting capability allows businesses to consistently enforce security and privacy policies, reduce overall security administration costs and ultimately increase employee productivity and customer satisfaction. Self-protection is also about recognizing and dealing with overload conditions that could jeopardize the integrity of the system.

Collectively these intuitive and collaborative characteristics can enable enterprises to operate efficiently with fewer human resources, helping decrease costs and enhancing a company's ability to react to change.

### An evolution, not a revolution

Delivering systemwide autonomic environments is an evolutionary process enabled by technology, but it is ultimately implemented by each enterprise through the adoption of these technologies and supporting processes. The path to autonomic computing can be thought of in five levels. These levels, defined below, start at basic and continue through managed, predictive, adaptive and finally autonomic.

| Basic<br>Level 1 | Managed<br>Level 2 | Predictive<br>Level 3 | Adaptive<br>Level 4 | Autonomic<br>Level 5 |
|---|---|---|---|---|
| Manual analysis and problem solving | Centralized tools, manual actions | Cross-reference correlation and guidance | System monitors, correlates and takes action | Dynamic business-policy-based management |

1. **Basic level**—a starting point of IT environments. Each infrastructure element is managed independently by IT professionals who set it up, monitor it and eventually replace it.

2. **Managed level**—systems management technologies can be used to collect information from disparate systems onto fewer consoles, reducing the time it takes for the administrator to collect and synthesize information as the IT environment becomes more complex.

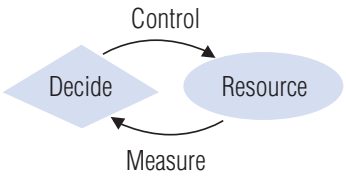
3. **Predictive level**—new technologies are introduced to provide correlation among several infrastructure elements. These elements can begin to recognize patterns, predict the optimal configuration and provide advice on what course of action the administrator should take.

4. **Adaptive level**—as these technologies improve and as people become more comfortable with the advice and predictive power of these systems, we can progress to the adaptive level, where the systems themselves can automatically take the right actions based on the information that is available to them and the knowledge of what is happening in the system.

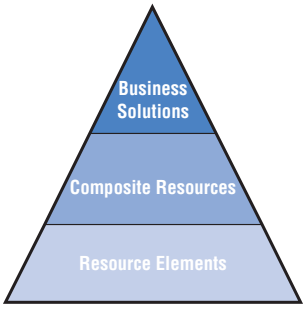
5. **Autonomic level**—the IT infrastructure operation is governed by business policies and objectives. Users interact with the autonomic technology to monitor the business processes, alter the objectives, or both.

### Autonomic computing architecture concepts

A standard set of functions and interactions govern the management of the IT system and its resources, including client, server, database manager or Web application server. This is represented by a control loop (shown in the diagram below) that acts as a manager of the resource through monitoring, analysis and taking action based on a set of policies.




These control loops, or managers, can communicate with each other in a peer-to-peer context and with higher-level managers. For example, a database system needs to work with the server, storage subsystem, storage management software, the Web server and other system elements to achieve a self-managing IT environment. The pyramid on page 3 represents the hierarchy in which autonomic computing technologies will operate.




The bottom layer of the pyramid consists of the resource elements of an enterprise—networks, servers, storage devices, applications, middleware and personal computers. Autonomic computing begins in the resource element layer, by enhancing individual components to configure, optimize, heal and protect themselves.

Moving up the pyramid, resource elements are grouped into composite resources, which begin to communicate with each other to create self-managing systems. This can be represented by a pool of servers that work together to dynamically adjust workload and configuration to meet certain performance and availability thresholds. It can also be represented by a combination of heterogeneous devices (databases, Web servers and storage subsystems) that work together to achieve performance and availability targets.

At the highest layer of the pyramid composite resources are tied to business solutions, such as a customer care system or an electronic auction system. True autonomic activity occurs at this level. The solution layer requires autonomic solutions to comprehend the optimal state of business processes—based on policies, schedules, service levels and so on—and drive the consequences of process optimization back down to the composite resources and even to individual elements.

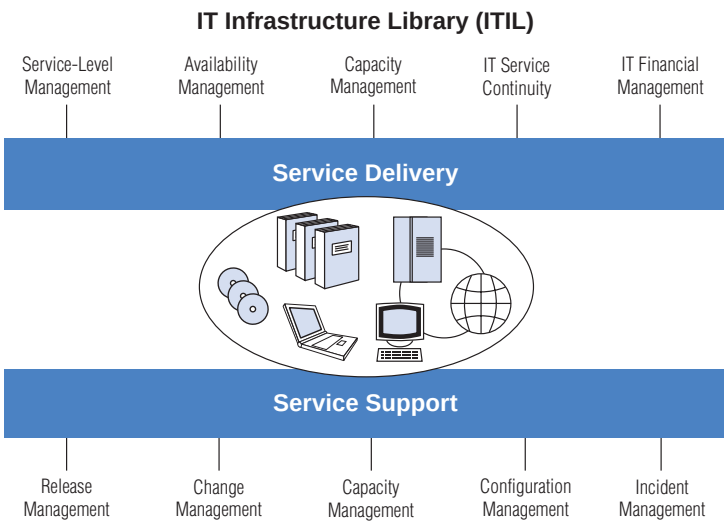
### Autonomic computing requires open standards

Many IT infrastructures have components supplied by different vendors. For multivendor components to participate in autonomic systems, there needs to be a set of standards for the managed elements' sensors and effectors and for the knowledge to be shared between autonomic managers that describe the interaction between the elements of an IT system. Some existing and emerging standards relevant to autonomic computing include:

- *Distributed Management Taskforce*
- *Common Information Model*
- *Internet Engineering Taskforce (Policy, Simple Network Management Protocol)*
- *Organization for the Advancement of Structured Information Standards (OASIS)*
- *Java™ Management Extensions*
- *Storage Networking Industry Association*
- *Open-grid systems architecture*
- *Web Services Security*

### How will self-management change the business of IT?

Small and large IT organizations perform similar sets of tasks to manage their systems. Because self-managing is about shifting the burden of managing systems from people to technologies, it is important to understand how self-managing capabilities impact the business of IT. The business of IT can be viewed as a collection of best practices and processes. For example, IT Infrastructure Library (ITIL) is a set of best practices and processes that IT organizations can use to deliver IT services to end users in a controlled, disciplined way. The ITIL service-management disciplines provide a framework that enables IT and end users to define their required levels of service performance (depicted in the diagram on page 4). ITIL leverages processes and tools to help realize high gains in efficiency—more than would be realized by improving either of those two entities alone.




### Implementing autonomic computing

Shifting the burden of managing systems to self-managing technologies does not happen overnight and cannot be solely accomplished by acquiring new products. Skills within the organization need to adapt, and processes need to change to create new benchmarks of success.

As companies progress through the five levels of autonomic computing, the processes, tools and benchmarks become increasingly sophisticated, and the skills requirement becomes more closely aligned with the business.

The basic level represents the starting point for many IT organizations. If IT organizations are formally measured, they are typically evaluated on the time required to finish major tasks and fix major problems. The IT organization is viewed as a cost center, with variable labor costs preferred over an investment in centrally coordinated systems management tools and processes.

In the managed level IT organizations are measured on the availability of their managed resources, their time to close trouble tickets in their problem management system and their time to complete formally tracked work requests. To improve on these measurements, IT organizations document their processes and continually improve them through manual feedback loops and adoption of best practices. IT organizations gain efficiency through consolidation of management tools to a set of strategic platforms and through a hierarchical problem management triage organization.

In the predictive level IT organizations are measured on the availability and performance of their business systems and their return on investment. To improve, IT organizations measure, manage and analyze transaction performance. The critical nature of the IT organization's role in business success is understood. Predictive tools are used to project future IT performance, and many tools make recommendations to improve future performance.

In the adaptive level IT resources are automatically provisioned and tuned to optimize transaction performance. Business policies, business priorities and service-level agreements guide the autonomic infrastructure behavior. IT organizations are measured on comprehensive business system response times (transaction performance), the degree of efficiency of the IT infrastructure and their ability to adapt to shifting workloads.

In the autonomic level IT organizations are measured on their ability to make the business successful. To improve business measurements they understand the financial metrics associated with e-business activities and supporting IT activities. Advanced modeling techniques are used to optimize e-business performance and quickly deploy newly optimized e-business solutions.

**Autonomic computing supports processes, tools, skills and benchmarks**

| Basic<br>Level 1 | Managed<br>Level 2 | Predictive<br>Level 3 | Adaptive<br>Level 4 | Autonomic<br>Level 5 |
|---|---|---|---|---|
| **Process**<br>Informal, reactive, manual | **Process**<br>Documented, improved over time, leverage of industry best practices, manual process to review IT performance | **Process**<br>Proactive, shorter approval cycle | **Process**<br>Automation of many resource management best practices and transaction management best practices, driven by service-level agreements | **Process**<br>All IT service management and IT resource management best practices are automated |
| **Tools**<br>Local, platform and product-specific | **Tools**<br>Consolidated resource management consoles, problem management system, automated software install, intrusion detection, load balancing | **Tools**<br>Role-based consoles with analysis and recommendations; product configuration advisors; realtime view of current and future IT performance; automation of some repetitive tasks; common knowledge base of inventory and dependency management | **Tools**<br>Policy management tools drive dynamic change based on resource-specific policies | **Tools**<br>Costing/financial analysis tools, business and IT modeling tools, tradeoff analysis; automation of some e-business management roles |
| **Skills**<br>Platform-specific, geographically dispersed with technology | **Skills**<br>Multiple platform skills, multiple management tool skills | **Skills**<br>Cross-platform system knowledge, IT workload management skills, some business-process knowledge | **Skills**<br>Service objectives and delivery per resource, analysis of impact on business objectives | **Skills**<br>e-business cost and benefit analysis, performance modeling, advanced use of financial tools for IT context |
| **Benchmarks**<br>Time to fix problems and finish tasks | **Benchmarks**<br>System availability, time to close trouble tickets and work requests | **Benchmarks**<br>Business system availability, service-level agreement attainment, customer satisfaction | **Benchmarks**<br>Business system response time, service-level agreement attainment, customer satisfaction, IT contribution to business success | **Benchmarks**<br>Business success, competitiveness of service-level agreement metrics, business responsiveness |

## Summary

Companies want and need to reduce their IT costs, simplify management of their IT resources, realize a fast return on their IT investment and provide high levels of availability, performance, security and asset utilization. Autonomic computing addresses these issues, and it is not just about new technology. This fundamental evolutionary shift in the way IT systems are managed will free the IT staff from detailed mundane tasks and allow them to focus on managing business processes. It can be accomplished through a combination of process changes, skills evolution, new technologies, architecture and open industry standards.

## To learn more

For information on the IBM autonomic strategy and integrated solutions from IBM, contact your IBM sales representative or visit **ibm.com**



IBM Corporation
Software Group
Route 100
Somers, NY 10589
U.S.A.

10-02



The IBM home page on the Internet can be found at **ibm.com**

G325-6904-00

# Classification of Research Areas in Autonomous Computing

## H/W or System

### 1. Project eLiza

Introduced in April 2001, Project eLiza is the name for an initiative that began in IBM's Server Group to drive autonomic capabilities into current products. It is the initiative to develop computers that need less human supervision because they fix themselves in the event of a failure, protect themselves from hacker attacks, configure themselves when adding new features, and optimize their CPU, storage and I/O resources to handle different levels of Internet traffic.

eLiza will give businesses the ability to manage systems and technology infrastructures that are hundreds of times more complex than those in existence today.

URL : http://www-1.ibm.com/servers/eserver/introducing/eliza/

### 2 Grid computing (Federated System)

Computing grids allow geographically distributed organizations to share applications, data and computing resources. A new model of computing, grids are clusters of servers joined together over the Internet, using protocols provided by the Globus open source community and other open standard/open source technologies such as Linux.

URL : http://www-1.ibm.com/grid/

**Ex 1) IBM to build world's most powerful computing grid**

A consortium of four U.S. research centers has picked IBM to build the world's most powerful computing grid, an interconnected series of Linux clusters capable of processing 13.6 trillion calculations a second. The grid system -- known as the Distributed Terascale Facility (DTF) -- will enable thousands of scientists around the country to share computing resources over the world's fastest research network in search of breakthroughs in life sciences, climate modeling and other critical disciplines. Funded by the National Science Foundation, the facility is a joint undertaking of the National Center for Supercomputing Applications, the San Diego Supercomputing Center, Argonne National Laboratory and the California Institute of Technology. The grid will include not only the fastest supercomputers, but high-resolution visualization environments, toolkits for grid computing and data storage facilities integrated into an information infrastructure called the "TeraGrid." The grid will

allow researchers to scan remote databases, run applications on far-flung computers and view complex computer simulations in real-time from widely separated locations. Unlike traditional supercomputers, which are typically housed at a single location, grids create vast pools of computing resources by connecting multiple, often widely-distributed supercomputers using the Internet or high-speed research networks as well as open source protocols from Globus. Organizations tap into the computing grids to access processing capacity, data storage and bandwidth in much the same way that consumers draw electricity from a power grid. IBM Global Services will deploy clusters of IBM @serverLinux systems at the four DTF sites beginning in the third quarter of 2002. The servers will contain the next generation of Intel's Itaniium microprocessor, code-named McKinley. IBM supercomputing software -- CSM and GPFS -- will handle cluster and file management tasks. Myricom's Myrinet interconnect will enable interprocessor communication. The system will have a storage capacity of more than 600 terabytes of data, or the equivalent of 146 million full-length novels. A substantial portion of the grid's storage infrastructure will be enabled by IBM TotalStorage products and technologies. The Linux clusters will be connected to each other via a superfast, 40 gigabit per second Qwest network, creating a single computing system able to process 13.6 teraflops. A teraflop is a trillion calculations per second. The DTF will be more than a thousand times faster than IBM's Deep Blue supercomputer, which defeated chess champion Garry Kasparov in 1997. Using open protocols, the Linux clusters will smoothly connect to a heterogeneous collection of existing high performance computers at the four labs, creating a giant virtual computer that may be accessed from any point on the Grid.

## 3. Oceano Project

The Océano project is designing and developing a pilot prototype of a scaleable, manageable infrastructure for a large scale "computing utility powerplant" that enables multi-customer hosting on a virtualized collection of hardware resources. A computing utility infrastructure consists of a "farm" of massively parallel, densely-packaged servers interconnected by high-speed, switched LANs. This project aims to address many of the open technical issues in these powerplant environments. Hosted customers increasingly require support for peak loads that are orders of magnitude larger than what they experience in their normal steady state. Thus, a hosting environment needs a faster turnaround time in adjusting the resources (bandwidth, servers, and storage), assigned to each customer to the dynamically fluctuating workload. The "colocation" hosting model uses dedicated infrastructure and servers for

each customer, typically in physical cages. In this model, enabling peak-load scale on demand requires large investments in standby, non-shared resources, which would be mostly underutilized and would occupy large amounts of physical space. Clearly, the colocation model is not suited to efficiently mitigate the differences between average and peak load. Océano introduces high levels of automation to dynamically adjust web sites to actual traffic demands over a massively parallel array of shared and distributed Linux servers

**The objectives of the Océano project include:**

- Implement an infrastructure that enables large numbers of hosted customers over Linux servers
- Reduce the costs of setting up and operating the hosting farms by automation
- Dynamically assign resources to accommodate planned and unplanned fluctuation of workloads
- Offer a wide variety of services levels to customers
- Secure sharing of resources across multiple customers
- Provide adequate reliability through massive redundancy, and automated re-provisioning

Océano will develop middleware and infrastructure, which provide composition of hosting services, including monitoring of Service Level Agreements, Dynamic Resource Allocation, and High Availability. This middleware and infrastructure will enable the development of powerplants that can handle multiple customer/applications and large surges in workload traffic.

## 4. HWLM: Heterogeneous WorkLoad Management

Workload management, a function of the OS/390* operating system base control program, allows installations to define business objectives for a clustered environment (Parallel Sysplex* in OS/390). This business policy is expressed in terms that relate to business goals and importance, rather than the internal controls used by the operating system. OS/390 ensures that system resources are assigned to achieve the specified business objectives.

This paper presents algorithms[iii] developed to simplify performance management, dynamically adjust computing resources, and balance work across parallel systems. It examines the types of data the algorithms require and the measurements that were devised to assess how well work is achieving customer-set goals. Two examples

demonstrate how the algorithms adjust system resource allocations to enable a smooth adaptation to changing processing conditions. To the customer, these algorithms provide a single-system image to manage competing workloads running across multiple systems.

**Ex 1) Adaptive algorithms for managing a distributed data processing workload**

☞ **See the Appendix I**

## 5. Storage Tank

The goal of the Storage Tank™ project is to provide a complete storage management solution in a heterogeneous, distributed environment. Storage Tank is designed to provide I/O performance that is comparable to that of file systems built on bus-attached, high-performance storage. In addition, it provides high availability, increased scalability and centralized, automated storage and data management.

Storage Tank uses Storage Area Network (SAN) technology that allows an enterprise to connect thousands of devices, such as client and server machines and mass storage subsystems, to a high-performance network. On a SAN, heterogeneous clients can access large volumes of data directly from storage devices using high-speed, low-latency connections. The Storage Tank implementation is currently built on a Fibre Channel network. However, it could also be built on any other high-speed network, such as Gigabit Ethernet (iSCSI), for which network-attached storage devices become available.

Storage Tank differs from conventional distributed file systems in that it uses a data-access model that requires clients to use servers to obtain only metadata, not the data itself. Rather, Storage Tank clients can access data directly from storage devices using the high-bandwidth provided by a Fibre Channel or other high-speed network. Direct data access eliminates server bottlenecks and provides the performance necessary for data-intensive applications.

The Storage Tank architecture also makes it possible to bring the benefits of system-managed storage (SMS) to distributed environments. Features such as policy-based allocation, volume management and file management have long been available on expensive mainframe systems. However, the infrastructure for such centralized, automated management has been lacking in the open-systems world. Storage Tank enforces policies and performs volume and file management without intervention from humans, thus making it an important step in the direction of autonomic computing.

## 6. Distributed Storage Tank

The Distributed Storage Tank project extends the Storage Tank™ system's application for heterogeneous and distributed file sharing, including file sharing in wide-area networks. The aim of this project is to allow a Storage Tank cluster to interoperate with other Storage Tank clusters as well as with various other types of file storage systems, such as NAS and Data Grids. Storage Tank servers in a local cluster are responsible for caching and replication management of remote files, which allows clients to access the data as if it were local. Distributed Storage Tank provides its clients with a unified file access interface to networked file storage systems with SAN-level performance and a single name space that is location transparent.

Our goal is to bring about a seamless interface for distributed and heterogeneous file storage systems that provides performance that is comparable to local file systems. Through cluster-managed shared caching, Distributed Storage Tank can also preserve the bandwidth of long-haul networks and reduce the average latency of file access because shared files need to be transported only once to each accessing site.

Grid Computing involves the sharing of a vast aggregation of geographically dispersed data and computing resources. Efficient and scalable storage management, high performance and easy access to geographically distributed data are often key requirements for Grid applications. Distributed Storage Tank extends Storage Tank's benefits of high performance, scalability and ease of management to geographically distributed data sharing.

Beyond the current focus of providing seamless and high-performance access to distributed files based on heterogeneous systems, the Distributed Storage Tank vision includes the incorporation of autonomic caching, replication and migration to facilitate optimized performance in distributed file sharing, while minimizing communication and storage overhead.

## 7. Autonomic Storage

The total cost of storage continues to increase due to the cost of storage servers, the rising complexity of storage management and the pervasive need for 24×7 operations. Autonomic storage is aimed at reducing the total cost of storage by providing systems that are largely self-managing, self-diagnostic and transparent to the user.

The Storage Systems group at IBM Almaden Research Center currently has several research projects in progress to address the complexities of autonomic storage. A key project in this area is Collective Intelligent Bricks (CIB).

CIB is a modular, scalable, storage controller built from simple, storage building blocks—called "bricks"—that allows scaling to petabytes of storage, while reducing the total cost of ownership (TCO) through self-management and a fail-in-place service strategy. Our CIB-Hardware project addresses the hardware aspect of the CIB system, and our CIB-Software project provides the software.

# Middleware

## 1. Intrusion detection/Security, BlueBox

Increasingly refined intrusion-detection techniques allow users to operate with confidence, in spite of the vast number of attacks that threaten computer systems. BlueBox is an intrusion prevention system, which creates an infrastructure for defining and enforcing very fine-grained process capabilities.

**Ex 1) Global Security Analysis Laboratory**

The research of the Zurich GSAL focuses on intrusion detection. Together with the Watson GSAL, the GSAL at IBM's Zurich Research Laboratory is the research center of competence for applied network security.

Our research work is dedicated to ensuring that the benefits and convenience of networked computing continue to outweigh the risks of operating in an open networked environment. Increasingly refined intrusion-detection techniques allow users to operate with confidence, in spite of the vast number of attacks that threaten computer systems.

The Zurich GSAL has acquired a worldwide reputation in the field of intrusion detection (ID). We support IBM's Managed Security Services and Security and Privacy services by developing methodologies and tools for the detection, prevention and analysis of security incidents.

**Ex 2) Internet Security Group**

Security and Privacy are core technologies for enabling the full exploitation of the Internet for e-business. The development and successful deployment of e-business applications requires carefully engineered and comprehensive security solutions. Such solutions must address all aspects of system security at the platform, operating system, network, application and infrastructure levels. This involves development of new cryptographic techniques and algorithms, their secure implementations, the design of secure networking protocols and operating environments and mechanisms to monitor and maintain overall system integrity. Such security solutions need to be standardised to provide/preserve inter-

operability and to ensure that these techniques are used in a correct way.

The Internet Security group is involved in all aspects of security as they pertain to the Internet. The group consists of Suresh Chari, Pau-Chen Cheng, Charanjit Jutla, J.R. Rao and Pankaj Rohatgi.

The group is working on several different projects in the Internet Security.

1. Side Channel Cryptanalysis: Power Analysis,
2. Electromagnetic Analysis
3. Partitioning Attacks on GSM Cards
4. Blue Box: A Host-Based Policy-Driven Intrusion Avoidance System
5. Privacy
6. Cryptography
7. Security for Pervasive Computing Applications
8. Virtual Private Networks (IPSec and Internet Key Exchange)
9. Secure Internet Multicast
10. Secure DNS

## 2. SMART - Self Managing and Resource Tuning DB2

IBM will be building a SMART (Self Managing and Resource Tuning) database into upcoming versions of DB2. This database is designed to reduce the human intervention needed to run and maintain a database. For example, the user can opt not to be involved and the database will automatically detect failures when they occur (and correct them) and configure itself by installing operating systems and data automatically to cope with the changing demands of e-business and the Internet.

The long-term vision is to offer companies the option of preventative maintenance or zero-administration/zero maintenance to reduce the total cost of ownership. LEO is one look at the future of "SMART" databases and how they will operate more effectively.

## 3. Gryphon

Gryphon is publish/subscribe middleware aimed at distributing large volumes of data in real-time to thousands of clients distributed throughout a large *public network*. A public network is a wide-area extranet or intranet that is too large or complex to be centrally administered to support specific applications. Such a network contrasts with well-controlled, centrally administered networks that have traditionally been the deployment environment of publish/subscribe systems such as TIBCO’s Rendezvous.

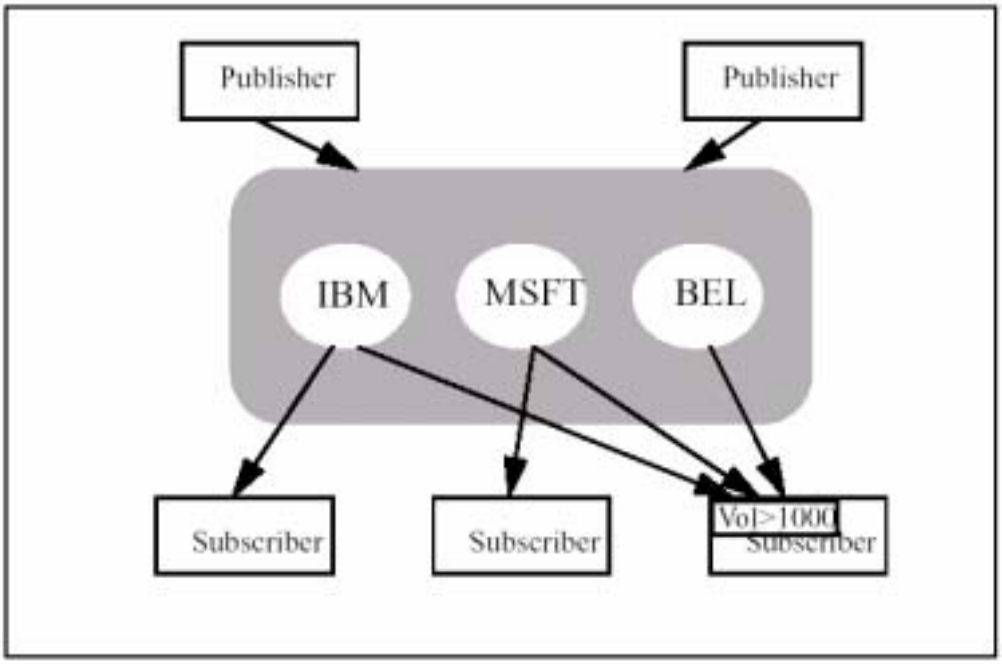
Gryphon has been deployed over the Internet for real-time sports score distribution at the US Tennis Open, Ryder Cup, and Australian Open, and for monitoring and statistics reporting at the Sydney Olympics.

**Technical Differentiators**

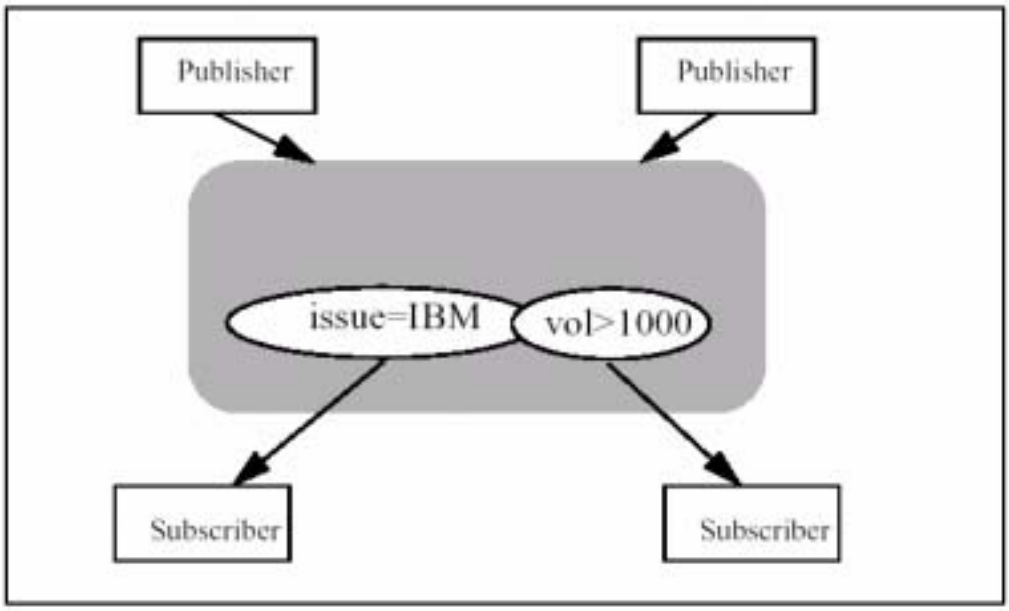
There are four areas of focus for Gryphon that distinguishes it from traditional publish/subscribe systems:

- Gryphon provides both topic-based and content-based publish/subscribe. Content-based subscription is the capability of a client to request messages based on their content. Topic-based addressing is an abstraction of numeric network addressing schemes. However, as applications evolve (Figure 1) and as new application are added, the addressing scheme must also change, requiring existing applications to be modified. With content-based subscription (Figure 2), delivery depends only on the content of messages, and therefore, applications can select different combinations of messages without changing an addressing structure.
- A publish/subscribe system deployed on a public network cannot depend on homogeneous router technology. In particular, reliance on LAN or IP multicast technologies is impractical in a large public network. Instead, universally adopted standards such as TCP/IP or HTTP must be used for all communication.
- A publish/subscribe system should be able to scale to support application growth. This includes supporting a large number of clients connecting to a single site and supporting linking of geographically distributed sites. As systems scale, the probability that a component will fail increases and a scalable publish/subscribe system must be able to recover from these failures.
- A publish/subscribe system deployed over a public network must provide security and privacy features to a degree not mandated over private secured networks. These features must include client authentication, access controls, and encryption/integrity of messages.

In Gryphon, functionality for these requirements is combined with a flexible information model supporting very large application deployment and application evolution. Below, we describe the features of Gryphon. Topic-based publish/subscribe systems limit application evolution. In this case, a subscriber interested in high-volume stock trades must subscribe to all topics and filter thousands of events per second at the client.




Content-based publish/subscribe systems scale with application growth.

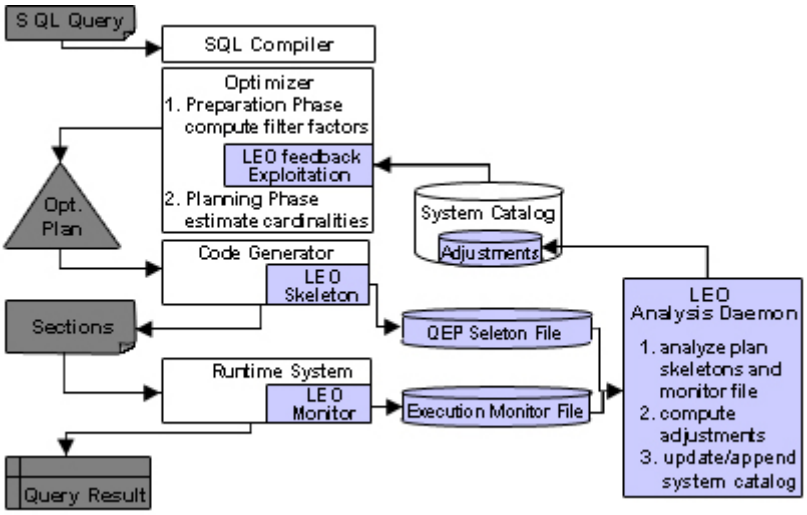



## Application / Utility

### 1. LEO

IBM Research's Learning Optimizer (LEO) project is the next-generation in query optimization for IBM's universal database. It minimizes administration and queries distributed, heterogeneous information. LEO is a equipped with a feed back loop, which

enhances the available information on a database. It is capable of learning from its past mistakes and fine-tunes queries as it learns about data relationships and user needs.




LEO: autonomic computing technology

## 2. Sabio

Sabio takes large collections of documents within an enterprise and breaks them down automatically into a taxonomy similar to that found on Yahoo! ®. This enables users to conduct advanced searches, locate specific expertise, etc. This technology is currently used in Lotus' Discovery server.

**EX 1) A New computer program classifies documents automatically**

**☞ See the Appendix II**

# Adaptive algorithms for managing a distributed data processing workload

by J. Aman, C. K. Eilert, D. Emmes, P. Yocom, and D. Dillenberger

**Workload management, a function of the OS/390* operating system base control program, allows installations to define business objectives for a clustered environment (Parallel Sysplex* in OS/390). This business policy is expressed in terms that relate to business goals and importance, rather than the internal controls used by the operating system. OS/390 ensures that system resources are assigned to achieve the specified business objectives. This paper presents algorithms developed to simplify performance management, dynamically adjust computing resources, and balance work across parallel systems. We examine the types of data the algorithms require and the measurements that were devised to assess how well work is achieving customer-set goals. Two examples demonstrate how the algorithms adjust system resource allocations to enable a smooth adaptation to changing processing conditions. To the customer, these algorithms provide a single-system image to manage competing workloads running across multiple systems.**

Although there has been an important role for a computing environment that has a single machine and a single copy or image of an operating system, a number of factors have converged to motivate use of multiple machines, especially when connected in a parallel fashion. These machines could be logical or physical configurations of what might otherwise be deemed a single machine, but each is controlled by a separate copy of an operating system.

Among the reasons for this interest in parallelism are:

- To increase total computing power available over a single image so as to reduce individual response time or to handle larger volumes of work, or both
- High availability due to the expectation that failure of any single component, at whatever level, will not cause the loss of all computing capabilities
- Access to lower-cost technology to use as building blocks for a larger system
- Ability to grow the total computing power in small increments to address needs as they arise with small incremental cost and no outage required

There is an obvious increase in complexity with the introduction of multiple images. It is natural to want to view them as cooperating and sharing resources. Considerable simplification results by seeing these multiple images as a single computing environment and having one set of

controls rather than separate controls for each image. System parameters previously requiring a human operator to monitor and set them are now controlled by the workload management (WLM) algorithms described in this paper. Workloads are dynamically balanced across images. WLM tracks those factors needed to best place incoming work and provides interfaces to make workload-balancing recommendations.

In a parallel environment, the WLM objective to simplify performance management while effectively using all computing resources poses a number of design problems that must be addressed. Given that some external controls are needed to reflect business goals and importance, but that low-level controls are not provided, the system must decide which resources to allocate to which work requests. It is up to the system to calculate how much of those resources to give and for how long a time. With due consideration for the danger of thrashing, it is up to the system to determine how often to make those changes and whether all the changes should be made at the same time.

With respect to the problem of balancing work across a parallel environment, the system must choose where to run each work request given the following constraints:

- Goals need to be achieved.
- Goals may not be known in advance.
- Resource requirements are unknown.
- Other work requests will be concurrently demanding resources in competition with new work requests that are also unknown.

Another problem addressed by the workload management algorithms is maximizing the use of resources across the parallel environment, especially where there are diverse machine sizes--the problem of configurational heterogeneity discussed in Reference 1. Finally, the underlying configuration must be concealed from end users and changes made transparent to them while allowing load balancing across equivalent servers [2].

In this paper, the next section describes related work in resource management and workload balancing. Then WLM concepts and the system model are described. The section following that one describes the WLM algorithms used for goal-oriented resource management. The two subsequent sections describe the WLM approach for balancing work across the parallel environment and the products that cooperate to realize the benefits of the WLM philosophy. The paper concludes with a summary of the current state of the art and some outstanding problems that are yet to be addressed.

## Related work

A number of alternatives [3] exist in deciding how system resources, such as access to the processor or processor storage, or both, should be allocated among eligible work. One possibility is that *no controls* are offered at all--the system queues and dispatches work automatically. The absence of external controls offers maximal simplicity and may well be adequate if the system is dedicated to a small number of work requests and is of sufficient capacity to handle all work quickly enough to satisfy the appropriate parties. This approach may also be sufficient if the system implements techniques to modify access to system resources as individual work requests "age," i.e., are observed to consume higher levels of resources. However, satisfaction with this approach will depend on how closely the system anticipates and implements the wishes or expectations of the end user(s) or installation in the absence of any external control. As the mixture of work in the system becomes increasingly diverse, with more complex human expectations on what should happen, the absence of any human control becomes less tenable.

A second approach is to *keep the system available for an "owner,"* thus protecting access of this special user to the system. This approach is more suitable to small systems but has a number of implementations. Condor [1] is a system that allows workstations to be used by others when idle but it checkpoints and preempts "foreign" work when the "owner" wants access to the machine. The Butler [1] system has a similar philosophy and will actually terminate "foreign" work when the "owner" wants access to the machine. Utopia, from the University of Toronto, also provides an option that allows the system to reject remote work when the "owner" needs the system back [1]. As a category, these implementations provide a limited partitioning of work as either "owner" or "nonowner," with no finer granularity for ranking work within or across these groupings.

A third approach is to *optimize system resources* so as to "keep the machine busy." Utopia [1] allows specification of a threshold beyond which "foreign" work is not accepted but otherwise is happy to offer service to all. This approach is an extension of the prior technique where a threshold of zero would be used for the amount of "nonowner" work that coexists with "owner" work.

A fourth approach is to *minimize response time*. This method is the implicit control in Reference 4. Although minimizing response time may seem desirable, it does not address conditions where not all work is equally important and misses the opportunity to make trade-offs to optimize some work at the expense of other work.

Once a wide variety of work requests can be in concurrent execution, it may not be sufficient to merely keep the machine busy. This probability suggests that the system administrator may want or need to control the priority of access to resources. One approach is to allow *specification of low-level "how to" performance controls,* exemplified by releases of MVS prior to Multiple Virtual Storage/Enterprise Systems Architecture System Product 5.1 (MVS/ESA* SP5.1) and by compatibility mode in MVS/ESA Version 5. Virtual Machine/Enterprise Systems Architecture* (VM/ESA*) [5] is a second operating system using this approach. Utopia also allows specification of priority controls [1].

The preceding paragraphs discussed how resources would be managed on behalf of work requests in the system. We now discuss alternatives for organizing multiple images on behalf of a workload. There seem to be two primary choices in this regard.

The first major scheme is to partition individual images into clusters, based on some attribute. One approach is to cluster images so that each cluster runs similar work or even the same "job." In the Scalable POWERparallel System 2 (SP2*), [6] some images function as server nodes, whereas others run individual work requests. Each node is separately configurable in terms of I/O, memory, and CPU capability. SP2 allows a system administrator to define separate pools of machines that are available to parallelize a particular job, run interactive users, or run nonparallelized jobs. Currently there is no support for time sharing or preemptive scheduling.

A second clustering approach is based on data affinity wherein each image is given ownership of a distinct set of persistent data (files, databases, etc.). The Tandem system and NCR 3600** system and Reference 4 all embody such an approach. The limitations of this approach are discussed in other papers [7].

The second major approach in organizing multiple images is one of shared data and shared work. For example, while Utopia assumes global file access [1], it uses geographic proximity (sometimes virtual proximity) to cluster images in the network. Specific resource requirements are kept in a system-provided file, which must be managed by system administrators, presumably with input from application owners who are aware of their own requirements. Reference 7 also assumes a data-sharing environment.

Other platforms need to assume that system capacity is configured for peak load, due to data affinity and the natural imbalance that will occur for realworld computing environments. This implies that those platforms require considerable excess capacity at off-peak times, which yields

substantial advantage to WLM where trade-offs can be made that reflect the intended use of computing resources according to business needs.

Once a parallel environment where multiple images are capable of handling a given work request exists, the question arises as to which image should be chosen. The decision as to where each work request should be placed and how to best choose the target image involves a number of trade-offs between what information is available and what resource management philosophy and controls are provided.

In the SP2 world, interactive users may be spread across nodes that are lightly loaded. Batch jobs may be submitted via IBM LoadLeveler* or NQS/MVS* (Network Queuing System/Multiple Virtual Storage), although parallel jobs may only be submitted by the former. LoadLeveler [2] attempts to balance work across a set of SP2 nodes by using:

- Job classes--Defined by the system administrator, jobs can be classified as short running, long running, etc.
- Job priority--How important a job is as defined by the owner's group, userid, and class. The priority of a job will determine whether LoadLeveler will schedule this job ahead of or behind existing queued jobs.

The LoadLeveler component--Interactive Session Support (ISS)--balances log-ins and application sessions across multiple servers based on factors such as link speed, number of connections, overall system load, and, optionally, machine speed. Although this is periodically reevaluated, there is no feedback to ensure that the recommendation reflects actual responsiveness.

Utopia performs load balancing under a dynamic algorithm that uses load indices for CPU queue lengths, free memory, disk I/O transfer rates, disk space, and number of concurrent users [1]. Other metrics may be used at the discretion of the installation, and applications are free to use their own metrics, although it seems that using different metrics would cause problems since different programs may be at cross-purposes in their routing approaches. A further challenge to Utopia's support is how to combine metrics into a single usable measure vs the more complex load vector proposed.

Utopia utilizes a "master" image to coordinate load data and in some schemes to make load decisions [1]. After placing each new work request, Utopia incorporates a load adjustment factor to account for latent demand. General resource demands are described in a system-provided file, though usually on an exception basis. It is unlike WLM, where resource demands are not

assumed to be known in advance. Utopia is intended to balance across potentially thousands of hosts, at which point the projected overhead is estimated to be 1 percent. With up to dozens of hosts, the overhead for balancing under Utopia is less than 0.5 percent.

Reference 4 assesses several alternatives to route work based on some knowledge of data access patterns and evaluates the sensitivity of the algorithms to incorrect information. The base algorithm against which all others are compared involves tracking where each transaction is routed, by class, and projecting what its expected response time will be on the basis of system parameters and static transaction attributes and then choosing the image that minimizes the expected response time. The paper shows that this algorithm is quite sensitive to these values, which is disconcerting in view of the practical difficulty of ascertaining these values and their tendency to change over time. The algorithm has a further tendency to overlook the cost of routing to an image that does not own the data used by the transaction.

The first alternative investigated in Reference 4 applies a threshold so that data affinity is enforced in routing unless the target image is overloaded, i.e., its projected response time is above the estimated optimal choice by a certain threshold percentage. The threshold approach is always superior to using data affinity as the sole determinant in routing. Under some conditions, using data affinity alone can cause the queues to become unbounded in length. However, choosing the best threshold is somewhat problematical since it must be sensitive to system utilization. WLM, by contrast, tracks the actual response time delivered with no assumptions on transaction attributes.

Reference 4 includes the interesting observations that optimization for a single work request can negatively affect overall results and that load balancing becomes more important as the overall load increases.

The second alternative investigated in Reference 4 assumes that transactions fall into either a short or long duration, and routes the former using the base algorithm, but routes the latter based on data affinity. This approach makes the further assumption that which category a given work request lies in can be readily determined at run time. The idea is to take advantage of idle capacity when the risk of making a mistake is low, but to force data affinity when the cost is high. This algorithm does better than the base algorithm and the first alternative, but the improvement is sensitive to utilization and communication costs. This alternative would require some sort of external specification by the system administrator, unlike WLM, which makes no assumption that the duration of a work request can be determined upon its arrival.

The adaptive approach discussed in Reference 4 uses feedback to adjust for incorrect information. This approach enhances the base algorithm by tracking actual response time values and uses this value to adjust the estimated response time formula.

Reference 7 uses lock contention in a shared-data environment as a technique to determine how to route work requests. In particular, groups of transactions that access the same data are routed to a given image to reduce the lock contention time. The basic objective is to ensure that each machine is kept at a "safe" utilization rate and to decide how to change the routing when any image is above its "safe" threshold. The algorithm depends on knowledge of factors such as:

- Threshold utilization
- CPU cost to process lock conflict when parties are on the same or different images
- Arrival rate of each transaction type and its CPU cost
- Affinity matrix representing the average number of times each transaction type waits for a lock held by each other type

### WLM system model

The complexity of specifying low-level controls to tune system resources leads to a natural desire to offer the system administrator the capability to specify goals for work in the system in business terms, rather than using low-level controls. The operative principle is that the system should be responsible for implementing resource allocation algorithms that allow these goals to be met. WLM is unique in offering externals that capture business importance and goals and implements them on behalf of the system administrator.

Two primary concepts and facilities that WLM provides need to be introduced at this point. The first is the ability to partition the universe of work requests into mutually disjoint groups, called *service classes*. This partitioning is called *classification* and is based on the attributes of an individual work request, which might include the userid that submitted the request, related accounting information, the transaction program to be invoked or the job to be submitted, the work environment or subsystem to which the request was directed, and so forth. Installations are able to specify which service class is associated with each work request by specifying the value for one or more attributes and the corresponding service class. Defaults and other techniques may be used to group work requests into each service class.

Each service class represents work requests with identical business performance objectives. To address the fundamental problem that the resource demands of most work requests are

unknown at the outset and can vary depending on parameters that may be known only at execution time, there is a need to allow the business objectives to change based on the resource demands of the work request. This is quite different from the requirement in other implementations that the resource demands be known in advance.

A service class is comprised of a sequence of *periods*, with a value defined by the installation to express how long a work request is considered to belong to each period. This "duration" is a measured amount of service consumed that incorporates time spent actually running instructions on a processor along with other components of service defined by the installation. Each work request starts in period 1 and is managed according to the first period goal (to be described in the next few paragraphs) until enough service is consumed to exceed the first period "duration." The work request is then moved to the second period and managed according to the second period goal, and so forth.

Each period has an associated *goal* and an associated *importance*, as alluded to above. Note that the durations may be assigned different values for distinct service classes, even when comparing the same period. In the same way, the goals for a given period in different service classes may be distinct. An installation may specify explicitly three major goal types for work requests. Certain activities associated with system work may be managed implicitly and are accorded special treatment and do not require installation specification. The goal types provided by WLM are *response time, discretionary*, and *velocity*. These types of goals are now described in turn.

Response time goals indicate a desire for internal elapsed time to be, at most, a certain value. "Internal" refers to the fact that the time is measured from the point where the work request is recognized by the system to the point where the work request is considered complete. Note that elapsed time refers to wall-clock time and, hence, includes delays when programs are not running on behalf of the work request. Use of wall-clock time is desirable since it reflects the impact on a user awaiting completion of the work request. The precise definition of when the clock starts or stops ticking to capture the elapsed time is documented for each particular environment and so is not elaborated in this paper.

The second goal type, discretionary, indicates that there is no business requirement for the work to complete within a certain predetermined elapsed time, and the system should use its discretion in giving resources to such work when it is ready to run. In an unconstrained environment, discretionary work will use available resources. In a constrained environment, discretionary work may be denied resources in favor of work requests with other goal types.

Optional controls not described in this paper allow the installation to ensure that discretionary work makes progress in a constrained environment.

The third goal type is velocity. Work requests that are not considered discretionary and do not have a set response time objective nevertheless may need further control to reflect the degree of delay that is tolerable once the work request becomes ready to run. Such work requests may be long-running (possibly "never-ending") and want to run periodically or intermittently, during which time the work request needs access to resources. Velocity goals address this category of work requests.

A final concept associated with periods, which was mentioned above, is that of *importance*. Importance is merely a relative ranking of work and is only a factor in constrained environments where the algorithms must make choices as to whose goals will be attended to first when system resources are reallocated. The algorithms attend to the goals of work at the highest importance before attending to those at lower importance levels.

The concept of period was introduced to demonstrate a fundamental behavior of WLM of work that addresses the variability of resource demands. WLM does not require the system administrator to know these demands in advance. Goals are allowed to change based on their cost. The term "period" is not used subsequently in order to avoid certain technical discussions and difficulties that are not central to the theme of this paper. The more general concept of "service class" will be used in the remainder of the paper. For a more complete description of WLM externals, please refer to Reference 8.

The WLM philosophy for resource adjustment is described in some detail in subsequent sections, but it is essentially a receiver-donor loop with respect to adjusting resources. The fundamental principle on which its success is based is that the system need not determine the optimal change at any given point. It is sufficient that the system makes an improvement when adjustments are made. This principle allows WLM to avoid the trap of over-analysis where system overhead may balloon in search of optimal solutions. By working only on a single problem at a time, the algorithms leave intact resource allocations that are working well.

With the description of how WLM addresses resource controls completed, the second major consideration is to describe what requirements and assumptions WLM makes in how the images of the parallel environment are organized. The section on related work described the two major approaches as clustering vs sharing.

WLM assumes that each image is potentially capable of running any application. Any configuration requirements are the responsibility of the installation. WLM requires no intervention to reconfigure the images based on workload so as to fully utilize capacity.

WLM is designed for a data-sharing environment. Specific resource requirements are not currently incorporated into WLM, e.g., configurations that are asymmetric with respect to devices, vector, or cryptographic facilities, etc. This asymmetry is currently assumed to be handled by subsystems or dynamically managed by operating system or subsystem cooperation. For example, certain routing techniques described in the section on balancing work across a parallel environment can be used to group servers that have identical data and facility access capabilities. These routing techniques include generic resource and sysplex routing and allow the installation to group like servers without WLM awareness of what their common capabilities might be.

The third consideration for WLM to address is the question of how to route work requests among the images of the parallel environment.

Reference 9 describes a number of approaches for work balancing, among which WLM can be described as an adaptive model. Static models are not sufficiently robust for commercial environments, given the expected variability in arrival rate, resource requirements, and so forth. The WLM structure possesses several desirable attributes described in the paper. First, it is important to ensure that the overhead associated with keeping the necessary data and the related calculations is low so as to avoid losing all advantage to extra system overhead. Second, the algorithms are not overly sensitive to inaccuracies in the data used to drive it. Third, simple approaches to load balancing prevail over complex algorithms. Finally, WLM will not move a work request that has already begun execution, since this is too expensive.

As will be discussed in subsequent sections, WLM also uses feedback to correct its view of how well each server is performing against actual business goals when deciding whether each server is a proper choice. However, WLM does not require nor use knowledge of data affinity in making its decisions. This is important for situations where this knowledge is unavailable or where the same cost is associated with accessing data from any candidate image as in a data-sharing environment. As noted in Reference 4, staying current on data affinity in the face of changing applications and usage patterns can make accuracy of data affinity assumptions problematical (and costly).

The WLM philosophy is to use actual measured results, which incorporate delays in all categories, and other indicators, without attempting to determine specific delays that cannot be directly controlled. Unlike Reference 7, which focuses on lock contention, WLM does not assume that data affinity can be determined on the basis of the attributes of an arriving work request. Of course, lock contention is not the only delay that must be considered in routing work requests.

The general philosophy adopted by WLM for balancing work across parallel systems is to place work where it has the "best" chance of meeting its goals, whatever they may be. This approach is superior to trying to fill up one machine prior to going to the next. It also addresses the problem of how to maximize use of resources across a parallel environment, especially where there are diverse machine sizes--the problem of configurational heterogeneity discussed in Reference 1.

The WLM design philosophy for routing consists of independent cooperating images with shared state data and uses a "push" model. A push model is one in which work requests are directed (pushed) to a given image for processing and is in contrast to a "pull" model wherein each image requests work explicitly. Unlike the approach described in Reference 9, WLM does not need to probe potential target images to see whether they are capable of absorbing new work as the shared state data are sufficient to make this determination. Note that in an OS/390* (formerly known as MVS) operating system environment, WLM can easily manage systems that are running at 90+ percent of capacity, whereas Reference 9 describes a model that works well in a range no higher than 70 to 80 percent of capacity. The approach of WLM is intended for dozens to hundreds of hosts, with overhead measured to be containable within 0.5 percent for several systems.

A number of benefits surface from the WLM philosophy of goal-oriented performance management. The most obvious of these benefits is the *simplification in defining performance objectives* and initialization states to the system. The system administrator is able to specify business objectives directly to the system in business terms. These objectives reflect both goals and business importance and apply to the entire parallel environment controlled by the business policy. It is still the responsibility of the system administrator to ensure that each service class contains work with similar goals, business importance, and resource requirements to acquire the maximum benefit from WLM. Placing work with similar goals but diverse resource requirements into the same service class will limit the ability of WLM to make effective resource trade-offs, to correctly project resource needs, and to project the effects of resource adjustments.

First, the system administrator does not have to understand low-level technical controls. There is no fiddling with dispatch controls. The system administrator does not have to understand trade-offs for setting dispatch priorities when a machine has a single very fast processing engine vs a single slower engine vs multiple slower engines vs multiple very fast engines. The system administrator does not have to individually set storage isolation targets (amount of processor storage that should be protected or restricted for a given address space), tune for the worst case, and then worry that the working set changes according to goal. The WLM algorithms will monitor and set the appropriate values automatically on behalf of the system administrator. *Effective use of capacity* is assured by the management algorithms.

The history of performance tuning has given rise to a number of heuristics to address different performance problem areas. Unfortunately, these "rules of thumb" are often wrong. For example, paging may be tolerated so long as goals are being met. In the past, the system administrator might set some control value, hear that users are unsatisfied, and then have to retune, all the time having to balance the needs of conflicting workloads. Performance tuning with WLM does not require that the system administrator readjust resources, a process that is iterative and expensive. Effective use of capacity is assured by the management algorithms.

Next, the system administrator does not have to worry about the placement of work to the best image and best server within the parallel environment. There is no requirement to define the resource requirements of work requests to the system. Effective use of capacity is assured by the management algorithms.

Finally, the business policy defined to WLM handles mixed workloads, e.g., interactive, batch, transaction processing, data mining environments, and so forth. The system is responsible for resource management of work in execution and for the management of delays and their impact on attaining goals. There is no need to partition the images or nodes of the parallel environment for each separate workload. The system administrator does not have to specify the resource demands of work in advance. Effective use of capacity is assured by the management algorithms.

The second major benefit of the WLM philosophy is to allow granular *growth* to be transparent to the installation. Transparency simplifies the problem of scaling the environment as the workload grows. WLM supports dynamic changes in adding or removing images, subsystems, and applications, as well as variability in workload characteristics and resource demands. Reconfiguration need not affect performance objectives. If there is insufficient capacity to meet all goals, the business policy determines the relative business importance in meeting each goal.

The addition of new applications need not cause revision of old objectives, with the attendant rebalancing of low-level controls. WLM dynamically adjusts to all these changes. WLM will also dynamically adjust to short-term changes, including spikes in demand.

The third major benefit is to support high *availability* objectives. This support includes rebalancing work when an image is removed and advising in the placement of restarting subsystem environments when their host system is removed. Change management is also simplified since a strategy of rippling hardware or software changes, or both, across images in the parallel environment while managing existing workload on remaining images allows for continuous operation 24 hours per day, seven days per week.

The fourth major benefit of the WLM philosophy is to require *no changes at the application level*. Support is provided by the operating system and major subsystem environments. It contrasts to an implementation such as Utopia [1], where there are no kernel changes, but some major applications are assumed to change to be sensitive to routing considerations.

## WLM algorithms for resource management

The Multisystem Goal-Driven Performance Controller (MGDPC) contains the resource management algorithms of WLM. The MGDPC is responsible for allocating computer system resources so that the customer's performance goals are met to the extent that the goals are achievable. The MGDPC must manage work across multiple systems. It must manage multiple types of work, from short transactions to processor-intensive batch transactions. It must manage client/server workloads, where resources must be allocated to servers to address the performance of the clients. It must manage workloads that vary, detecting performance problems and reallocating resources. It must manage multiple resources. And it must do all of this efficiently. The MGDPC must act like a very good systems programmer. The following subsections describe how it is done.

The code in the MGDPC combines the performance management approaches of an experienced systems programmer with analytic algorithms. The systems programmer in the MGDPC has the advantages of a wealth of data, analytic algorithms that run at machine speeds, the opportunity to make resource changes every ten seconds, and updated data and feedback on previous decisions every ten seconds. The MGDPC can be thought of as a data collection and analysis system, resource adjustment, and feedback loop extending across a set of interconnected, cooperating, independent computer systems.

The MGDPC collects performance data, measures the achievement of goals, selects the service classes that need their performance improved, selects bottleneck resources, selects donors of the resources, assesses the impact of making resource reallocations, and makes the reallocations if there is a net benefit to the changes. The MGDPC is invoked once every ten seconds, referred to as a policy interval, performing detection and correction of actual or anticipated performance problems so as to make the operating system adaptive and self-tuning.

**Independent and cooperating.** The MGDPC is responsible for managing the performance of a workload that is distributed across a set of interconnected, cooperating, independent computer systems. These computer systems are said to be cooperating in the sense that each is exchanging operational measurement data with the other computer systems in the set. They are said to be independent in the sense that each is an entirely separate, wholly functional computer system whose resources are controlled by its own copy of the operating system. Each system operates independently and considers itself the local system. To each system, the remote systems are all the other systems being managed. Each system considers itself local and all other systems remote. The MGDPC is implemented as distributed intelligence. No system considers itself the master.

The primary objective of the MGDPC is to meet performance goals across all the systems being managed. This objective is met without any centralized control. Instead, each system receives performance data from all the other systems being managed and, based on its view of how the entire distributed workload is doing, makes resource allocation decisions to best meet sysplex (System/390* Parallel Sysplex* [10])-wide goals. A secondary objective of the MGDPC is to meet performance goals on its local system, in which case resource allocation decisions are made using local and remote data.

Each local MGDPC collects data on its local system, periodically broadcasts its view to the other systems in the sysplex, and implements mechanisms that can run independently on each system so that each system knows which class of work to help, by how much, and in what order, and knows the effects that resource reallocations on the local system will have on the sysplex performance of each class of work.

Each system's understanding of the sysplex effects of resource reallocations is the key to each system being able to independently make local resource trade-offs to achieve sysplex performance goals. Each system must also understand which portion of the problem it must solve so multiple systems do not all try to solve all parts of the problem at the same time.

Another feature of the MGDPC allows the systems to reallocate resources to help work that is doing poorly on the local system even though the work is doing well from a sysplex perspective. This local optimization is allowed as long as it does not adversely affect the relative sysplex performance of other classes of work. If an individual system determines that there is nothing it must do to assist work to achieve sysplex performance goals, it is free to work on local performance problems to the extent that sysplex goals are not adversely affected. It has enough data to project the effect of local resource reallocations on sysplex goals.

**Fundamental concepts.** In this subsection, the fundamental concepts of MGDPC operation are discussed.

*Data histories*. The MGDPC algorithms require efficient access to large quantities and varieties of performance data. Individual MGDPC algorithms need data summarized over different periods of time. Since individual algorithms also need different levels of statistical confidence in the data, they need to be able to look at different minimum numbers of data points. The use of the data determines how far back in time it is necessary to look or the minimum number of data points required to get a valid representation of a phenomena, or both. It is therefore important to maintain the number of data points represented in the performance data, and it is not always sufficient to merely keep a single summary value. Keeping all the individual observations of all the types of performance data in virtual storage, and searching and summarizing on demand, would consume far too much storage. Accessing the data from disk would require far too much time.

The MGDPC solved the problem with data histories. A data history is a mechanism to collect and analyze data over time. By using data histories the MGDPC can use data that have enough samples to be representative without using data so old that the data might be out of date. A data history contains *n* rows of data and a roll counter that determines when data should roll out of each row. Each row represents data from a range of time in history. Row 1 contains data from the most recent period only. Subsequent rows contain varying ranges of older data. Values for the number of rows have been found that have been proven to be effective for the OS/390 environment. The roll counter controls when to roll a row of data from one time range to another further back in time. The roll counter is incremented each policy interval. Each row has associated with it a number that corresponds to the roll counter value specifying when the data in the row should be rolled into the next row. If the counter value of row *m* is 1, it means row *m* is rolled into row *m* + 1 every interval. If the counter of row *m* is 4, it means row *m* is rolled every fourth interval.

Data are added to the history as follows. New data are added to row 1. At the end of each policy interval the oldest row whose roll counter value evenly divides into the current roll counter value is found. The content of that row is added to the next numerically higher row. The content of all the numerically lower rows are moved up one row, leaving row 1 empty. When it is time to roll data out of the last row in the history, the data are discarded. To obtain data from a data history, the data from rows 1 through *p* are added together. The value of *p* is chosen such that the data used were gathered over a long enough interval with enough samples to be representative.

Given the ability to summarize data for varying ranges of time by simply including data from different rows of the history, data can be summarized for a minimum time or a minimum number of observations, or a combination of these criteria. The MGDPC uses the data history facility extensively. Histories are used for state samples, response time distributions, processor consumption, performance index calculations, service consumption per transaction, server topology determination, and other purposes.

*Performance index*. A fundamental problem with trying to meet performance goals and make trade-offs among different work with different goals is knowing how work is doing relative to its goals and relative to other work. The solution used by the MGDPC is the performance index. The calculation of the performance index for a class with a response time goal is:

performance_index = actual_response_time / goal_response_time

It is a calculated measure of how well work is meeting its defined performance goals. The performance index allows comparisons between work with different goals. A performance index of 1.0 indicates the class is exactly meeting its goal. A performance index greater than 1.0 indicates the class is performing worse than its goal, and a performance index less than 1.0 indicates the class is performing better than its goal.

New performance indexes are calculated for every policy interval. Performance indexes are calculated from enough recent completions to be representative of the results for the class. Both sysplex and local performance indexes are calculated for each class on each system. To operate independently, each system must have enough information to be able to calculate a performance index for each class. To provide this information, each system sends updated information to all the other systems every policy interval. The information is stored in two histories. Local information is stored in a local history, and data from the remote systems are stored in a history for data from remote systems.

A projected response time is calculated for each in-flight work unit. The projected response times for in-flight work are combined with data from the actual response completions to calculate the performance index. The local performance index represents the performance of work units associated with the class on the local system. The local performance index is calculated from data from the local response time history. The sysplex performance index represents the performance of work units associated with the class, across all the systems being managed. Each system independently combines the local and remote data histories to compute a sysplex performance index.

*State sampling*. The first action to be taken when trying to solve the performance problem of a service class is finding out what the problem is. The MGDPC must determine why the work is being delayed. Many delays can be measured quite precisely, but the cost is prohibitive. The MGDPC solved this problem with state sampling. Four times a second, the MGDPC samples every work unit in every system being managed. Four times was chosen as a value because it is frequent enough but not prohibitive in cost. From these samples, the MGDPC builds a picture of the work in each class. It learns where each class is spending its time. It learns how much each class is using each resource and how much each class is delayed waiting for each resource. The samples are aggregated for each policy interval, and from this picture of the work in each class, the MGDPC can determine what to do. The state sampling implemented by the MGDPC is very efficient, requiring not more than one percent of the processor time to accomplish its task. The cost of state sampling is by far the largest contributor to system overhead among the various functions performed. **Figure 1** shows the types of state samples.

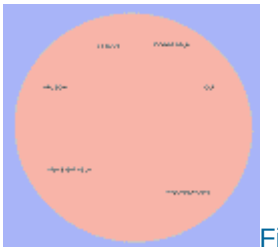


Figure 1

*Server topology*. Client/server workloads introduce a further level of complexity into managing resources to meet performance goals. The client service classes [11] have the performance goals but are served by one or more server address spaces. The client service classes do not consume computer system resources. The resources are consumed by the server address spaces serving the client service classes. So computer system resources must be allocated to the server address spaces to meet the goals of the client service classes. The MGDPC must understand the client/server relationships and must be able to project the effects on the client

service classes of making resource adjustments to the server address spaces. The MGDPC must be able to project second-level effects.

The client service classes in the diagram of **Figure 2** are labeled CICSA and CICSB (from Customer Information Control System, CICS*). Work requests classified to CICSA and CICSB receive service from several server address spaces. CICSA is served by server spaces TOR1, AOR1, and AOR2. CICSB is served by server spaces TOR1, AOR2, and AOR3. Achieving the goals of CICSA and CICSB requires that adequate computer system resources be allocated to the server address spaces--TOR1, AOR1, AOR2, and AOR3--since resources cannot be directly attributed to or allocated to CICSA and CICSB.

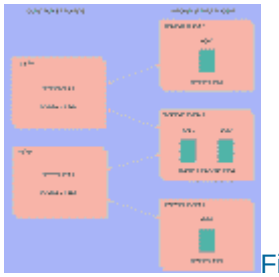


Figure 2

The problem of learning the client/server relationships was solved by sampling. The problem of allocating computer system resources to server address spaces to meet the goals of the client service classes was solved by dynamically creating internally defined server service classes and assigning the server address spaces to them based on the client service classes they were observed serving. The problem of projecting second-level effects was solved using a proportional aggregate speed algorithm.

Four times a second, the MGDPC samples control blocks set by the server address spaces to detect which client service classes are being served. From these samples, the MGDPC learns which server address spaces serve which client service classes and in what proportion. The MGDPC reevaluates these client/server relationships once a minute so the topology built will reflect changing client/server relationships. Server address spaces are also moved among internal service classes once a minute to reflect any changes in the client/server topology.

For each distinct combination of client service classes observed being served by one or more servers, an internally defined server service class is dynamically created. In the example in Figure 2, these combinations are (CICSA), (CICSA, CICSB), and (CICSB). AOR1 serves only CICSA. TOR1 and AOR2 serve both CICSA and CICSB. AOR3 serves only CICSB. On the basis of these combinations, the MGDPC creates the corresponding internal server service

classes 1, 2, and 3 and moves TOR1, AOR1, AOR2, and AOR3 to them for management. Internal classes are a mechanism for collecting data on and managing servers to meet the goals of clients. To meet the client service class goals of CICSA and CICSB, the server address spaces will be managed by managing server service classes 1, 2, and 3.

Computer system resources are allocated to these internal server service classes in order to meet the performance goals of the client service classes. The topology represents the client/server relationships and the proportion of time each server is serving each client. This learned information will adapt over time, because the relationship between clients and server address spaces is dynamic. The server topology samples are kept in a history. The history mechanisms slowly age the samples out so there are less likely to be abrupt changes based on short-term effects.

*Proportional aggregate speed*. In the client/server case, the MGDPC must improve the performance of the client service classes indirectly. The MGDPC must be able to assess the effect on a client service class, e.g., CICSA, from improving the performance of an internal service class, e.g., Internal Class 2. This improvement is proportional to the extent to which the client service class, e.g., CICSA, is served by the server spaces in the internal service class. To be able to project the effects on clients of the resources allocated to the servers, the concept of the proportional aggregate speed of a client class was introduced.

For a nonserved class, speed is defined as the classes' processor "using samples" [12] divided by all of the nonidle samples of the class, multiplied by 100, and results in this calculation:

speed = processor_using_samples / nonidle_samples × 100

If the work units in the class were never delayed, the speed of the class would be 100.

The proportional aggregate speed of a client service class is the apportioned speed of all the internally defined server service classes serving it. The proportional aggregate speed for each client service class is determined by allocating all of the client's server's state samples to the client service class in proportion to the portion of time that each server service class was observed serving each client service class. The portion of time is determined from the client/server topology. The proportional aggregate speed of a client service class is calculated by dividing the total processor using samples apportioned to the client service class from all server service classes, divided by the total processor using samples plus all delay samples apportioned to the client service class from all server service classes. The calculation follows:

$$\frac{\sum_{\text{servers}} \text{processor using samples apportioned to class A}}{\sum_{\text{servers}} \text{using and delay samples apportioned to class A}}$$

For each client service class, the client's performance index is plotted versus the proportional aggregate speed of the client class. This plot, shown in **Figure 3**, is then used to determine the effect, i.e., the performance index delta, on the client of changing the allocation of system resources to server address spaces.

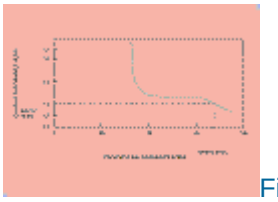


Figure 3

*Performance index delta*. Just as the performance index is the measure of how well a class is doing with respect to its goals, the performance index delta is the common unit of measure for the relative value of making resource reallocations. The performance index delta is always calculated from delay sample deltas. Each individual resource fix algorithm uses algorithms unique to the resource to determine the delay sample delta that will result from a resource reallocation. Then the delay sample deltas are used to calculate the performance index deltas that are used to assess the relative value of the resource reallocation.

For nonserved classes, performance index deltas are calculated as shown below. The calculation is a three- step process. First, the projected response time delta is calculated. It is the actual response time multiplied by the proportion of the total nonidle samples represented by the sample delta. If the total samples were 100, and the delay samples projected to be eliminated were 20, the response time would be projected to be reduced by 20 percent. Then the delta to the local performance index is calculated from the projected response time delta. Finally, the sysplex performance index delta is calculated from the fraction of total observations in which the class was observed on the local system. Note that these equations apply to both receivers and donors. For a receiver, the delay sample delta is negative, so the performance index is projected to be lower, which is an improvement. For a donor, the delay sample delta is positive, reflecting additional delay and an increased performance index.

proj_response_time_delta
= delay_sample_delta / nonidle_samples × actual_response_time (1)

local_proj_performance_index_delta

= proj_response_time_delta / goal (2)

sysplex_proj_performance_index_delta

= local_observations / sysplex_observations

× local_proj_performance_index_delta (3)

For client/server classes, the performance index delta is determined from the client's proportional aggregate speed plot. To read the projected performance index from the plot, a projected proportional aggregate speed must be calculated. The calculation starts with delay sample deltas calculated by the individual fix algorithms. Projected delay sample deltas are calculated for each server that serves the client class. Then the sample deltas are apportioned to the client class based on the server topology. The server topology represents the client/server relationships and the proportion of time each server is serving each client class. After the sample deltas of the server are apportioned to the client, the projected proportional aggregate speed is calculated for the client class. Then, the projected performance index is read from the client's proportional aggregate speed plot. The performance index delta of the client class is the difference between the projected performance index of the client class and the current actual performance index of the client class. Proportional aggregate speed plots contain sysplex data, so no local-to-sysplex performance index delta conversion is required.

**Policy adjustment framework.** The policy adjustment algorithm is invoked periodically to assess reallocating system resources to better meet performance goals. The policy adjustment algorithm is invoked every ten seconds. Ten seconds was chosen as a value sufficiently small to be responsive to changing system conditions and external user perceptions, but sufficiently large to allow enough samples to be acquired on which to base new resource allocations. This period of ten seconds is referred to as a policy interval. The effects of the resource reallocations made during one policy interval are observed in subsequent policy intervals and function as a feedback loop for continuous adaptive policy adjustment.

The resource readjustment actions taken are incremental, having the advantage of leaving resource allocations alone except when changes are needed to meet performance goals. Since

the MGDPC is invoked every ten seconds, there are ample opportunities for it to make sufficient changes to address any problems and to obtain feedback before making further changes. Some of the most "human" behavior observed in the MGDPC is its inclination to jump in immediately to help whenever it can but also to recognize when its help is not needed.

The MGDPC helps by searching for the one set of actions most beneficial to the service class most in need of help. The *select receiver* algorithm is used to select the receiver service class most in need of help and to select alternative receivers if needed. The *find bottleneck* algorithm is used to find the resources causing the receiver delay. The *select donor* algorithm selects potential donor service classes to donate bottleneck resources to the receiver. The *net value* algorithm determines whether there is net value to the donation. The *receiver value* algorithm determines whether there is sufficient value to the receiver to make the donation worth doing. The *fix delay* algorithms are unique for each resource and are used to assess changes and calculate the value of changes in common value units (performance index deltas) to be used by net value and receiver value algorithms. These algorithms are invoked in a loop, referred to as the *policy adjustment loop*, illustrated in **Figure 4** until one receiver service class is helped or all service classes have been assessed, and there is no way or no need to help. All of these algorithms are discussed further in the following subsections.


Figure 4

The policy adjustment loop selects a class to help (select receiver), determines the resource causing the class the largest delay (find bottleneck), assesses reallocating resources from one or more donor classes to the receiver (fix delay, select donor, net value, and receiver value), and makes the changes if there is value to the aggregate attainment of goals. If there is insufficient net value or receiver value with one set of donors, other sets of donors will be assessed. If there is insufficient net value or receiver value with any combination of donors for a given resource and receiver, the resource causing the receiver the next largest delay will be determined and donors of that resource assessed. If there is no combination of donors of any resource for a given receiver, the next most deserving receiver will be selected, and all resources and donors assessed for that receiver until all possible receivers have been

assessed or until a receiver is helped. When a receiver has been helped, the MGDPC exits to await feedback on the changes during the next policy interval.

The policy adjustment loop and the select receiver, find bottleneck, select donor, net value, and receiver value algorithms are all common for all the resources. The fix delay algorithm is unique for each resource. This loop is a very powerful framework for performance management. A fix algorithm for any resource can fit into this framework. The only requirements are that a delay that indicated a lack of the resource can be sampled, a control variable controlling access to the resource can be defined, and a relationship can be found between the control variable and the resulting delay samples. These concepts, as they apply to dispatch priority, I/O priority, storage allocation, and MPL [13] slots, are described in later subsections.

*Assemble performance data*. At the beginning of each policy interval, performance data that have been collected asynchronously by state sampling and other processes are assembled into efficiently accessible data structures to prepare for running the adjustment algorithms. Performance indexes are calculated, data received from other systems are assembled into histories, points are added to plots, sample sets are built, and the server topology is updated. It is similar to what a system programmer would do in preparation for tuning a system. The difference is that the MGDPC does data assembly at machine speed.

*Select receiver*. The first decision the policy adjustment algorithm must make is to decide which class to help. The MGDPC makes incremental improvements every ten seconds. It attempts to find one receiver to help each policy interval and looks for the most deserving receiver each time. Making incremental changes ensures that there is a solid base of feedback data to use in the algorithms during each policy interval. Potential receivers are selected based on importance, sysplex and local performance index, and the likelihood of the MGDPC being able to help the receiver. Classes that are missing sysplex goals are selected before classes that are meeting sysplex goals but missing goals on the local system. Classes missing goals are selected in order of importance. Classes meeting goals are selected in sysplex performance index order and then in local performance index order. Because the worst-off classes are selected first, it is more likely that a resource reallocation with significant value will be found.

The policy adjustment algorithm also remembers whether it has tried unsuccessfully to help a receiver in a recent interval. If it did, the select receiver algorithm skips over assessing the receiver. This is an optimization which saves the cycles that would be used to again come to the conclusion that the receiver could not be helped. Select receiver also knows when to leave resource allocations alone. It only selects classes that have a current performance index above

0.9. Classes that are meeting goals but have a performance index above 0.9 are close enough to going over 1.0 to merit some attention if their performance can be improved without harming other work. However, classes with a current performance index of 0.9 or lower are easily meeting their goals and do not need help. The select receiver algorithm has the intelligence to know when to quit.

*Find bottleneck*. Once the receiver class has been selected, the next step is to select which resource delays to address. For nonserved classes, the selection of the next bottleneck to address is made by selecting the delay type with the largest number of delay samples that has not already been selected for this receiver during the current policy interval. If fixing that delay does not provide sufficient receiver or net value, the next largest delay is assessed and so on until all delays have been considered.

In the client/server case, both a bottleneck resource and the associated bottleneck server must be selected. The selection of which bottleneck to address is made by selecting the server-delay combination with the largest number of apportioned delay samples that has not already been selected during the policy interval. The server samples are apportioned to each client class on the basis of the server topology described previously. The delay type having the largest number of samples apportioned to the receiver class is selected as the resource bottleneck delay type to be addressed on behalf of the receiver class. The server that experienced the bottleneck delay is selected as the bottleneck server.

In either the nonserver case or the client/server case, on each invocation, the delay with the next largest number of delay samples is selected to be assessed. No minimum number of samples is required for a delay to be assessed for fixing. Any defined minimum would by its nature be arbitrary and might eliminate a valuable change from consideration. The MGDPC handles the problem of making insignificant changes by requiring sufficient receiver value for a change. If too few samples would be eliminated to make a significant improvement, the change for that delay would fail the receiver value algorithms. But at that point the decision would have been well thought out, not arbitrary.

*Generic delay fix*. There is a specific fix algorithm for each delay addressed by the MGDPC. The function of each fix algorithm is to improve the performance of the receiver class or determine that there is not sufficient value to make a change. Improving performance is done by changing a control variable specific to the delay being addressed. To determine value, the fix algorithm must be able to project the performance index delta that results from changing the control

variable. A fix algorithm specific to the delay to be addressed is invoked when that delay is selected by the find bottleneck algorithm.

Each fix algorithm is responsible for selecting potential donors of the resource, projecting the effect on attainment of performance goals if the donor or donors donated to the receiver, accepting or rejecting changes, selecting alternate donors, and reallocating the resources if any reallocation is found that has net value. In all cases, the individual resource fix algorithm projects delay sample deltas and uses them to project performance index deltas for the receiver and donor or donors. The projected performance index deltas are then used to determine whether the resource reallocation has net value. The details of these calculations are specific to individual resources and are described later.

*Select donor*. The purpose of the select donor algorithm is to choose the most eligible class that will donate the required resource to the receiver from the set of classes owning that resource. Donors are selected in an order that is generally the reverse of the order used to select receivers. However, the donor order is dynamic even within a policy interval. Multiple donors may be needed to provide enough of a resource donation to reach sufficient receiver value. As each tentative donation is evaluated and accumulated, the resulting performance index changes are calculated and factored back into the donor order. The dynamically changing list feature is important, especially when finding storage donors, where donation can take many forms.

Additional constraints on the select donor algorithm require that the donor own the resource needed by the receiver. For example, a dispatch priority donor must be running at a dispatch priority that is at least equal to the dispatch priority of the receiver. In the case of storage, the donor can hold the resource in any form. For example, if the receiver needs MPL slots, the donor does not have to donate MPL slots. What the receiver actually needs is storage for MPL slots. The donor's storage can be in the form of a protective processor storage target or in the form of MPL slots. If the resource required to help the receiver is increased I/O priority, the I/O donor must be in the same I/O cluster [14] as the I/O receiver. This requirement must be met for the donation to have any effect on the receiver.

In addition, the select donor algorithm will not select a class as a donor of a resource if it was selected as a receiver for the same resource in the same policy interval. This is an example of including the experience of a system performance analyst in the code.

*Net value*. The net value algorithm keeps the MGDPC from making bad resource reallocations. The performance index value for a class is the measure of how well that class is meeting its

specified goal. The measure of the value of a contemplated resource reallocation to the receiver is the projected change in the performance index of the receiver that occurs as a result of the contemplated resource reallocation. Similarly, the measure of the net value of a contemplated resource reallocation is the improvement in the performance index of the receiver relative to the degradation of the performance index of the donor.

The net value algorithm uses the projected performance index deltas for the receiver and donor to calculate whether there is net value to the contemplated donation from the donor to the receiver. Net value takes the sysplex and local performance indexes into consideration as well as the importance of the receiver and donor or donors. All donors are checked. A receiver will only be improved by reallocating resource from a specific donor if a net positive value to the resource reallocation is projected. If using a donor to improve a receiver is projected to result in more harm to the donor than improvement to the receiver relative to the goals and importance, the resource reallocation is not done. If the result will yield more improvement for the receiver than harm to the donor relative to the goals, the resource reallocation is done.

*Receiver value*. The receiver value algorithm is a key feature that keeps the MGDPC from making resource reallocations that are either too small or too drastic. A receiver will only be helped when sufficient receiver value is projected. The receiver value criteria are a minimum performance index improvement or the elimination of a minimum number of delay samples. These criteria are designed to reject very small improvements. The reason for rejecting actions having too little receiver value is to avoid making changes that yield only marginal improvements. Marginal changes are not made, and the MGDPC goes on to select and assess another bottleneck for the current receiver or to select a new receiver.

The receiver value criteria also perform the function of indicating to the "individual resource delay fix algorithm" at what point it has given the receiver enough help. These criteria keep one system in a sysplex from trying to solve all of the performance problems of a class when the class is running on more than one system. The criteria also keep multiple systems in the sysplex from trying to solve all parts of the problem simultaneously and running the risk of making too much of a correction. None of the systems require explicit communication or coordination to know how much of the problem is theirs to fix.

*Send data*. At the end of each policy interval on each system, the MGDPC sends data to all the other systems in the set of independent cooperating systems being managed. Performance data and control data are sent. This action is of key importance to the distributed intelligence of the MGDPC.

The MGDPC on each system maintains a history for each type of performance data received. The histories cover enough intervals of time such that late or out-of-order data do not require special handling or error processing. The late data just roll into the history whenever the data arrive. If a system fails, and its data stop arriving, it simply stops being included in the history, and stops being considered in decisions. The data from the failing system will gracefully age out of the history without the other systems having to be specifically notified that a system went down. It eliminates the need for special-case and error-handling mechanisms and abrupt changes in resource allocation policies on individual systems. The use of histories to manage the remote performance data allows the systems being managed to operate independently.

Control data are also sent to remote systems at the end of each policy interval. An example is sending the fact that an I/O priority change was made. I/O priority changes require a relatively longer time to provide feedback than other changes such as dispatch priority. Since these changes take longer to provide feedback, they are made less frequently. To accomplish this longer interval between changes, each system must know whenever another system made such a change.

**Processor delay fix.** This subsection describes how performance is improved by reducing the delay the receiver experiences waiting to run on the processor. The controlled variable in this case is the dispatch priority.

*Theory*. The processor delay experienced by the receiver is a function of the processor time available to the receiver. Processor time available to the receiver is a function of the processor demand from work running at higher dispatch priorities than the receiver and the processor demand from work running at the same dispatch priority as the receiver. Processor delay is also a function of both the receiver's mean-time-to-wait and the receiver's mean-time-to-wait compared with the mean-time-to-wait of the other work at the same dispatch priority as the receiver.

For the processor delay fix algorithm to fit with the resource adjustment framework discussed previously, the processor delay fix algorithm has to be able to project the processor delay sample deltas that would result from dispatch priority changes. Multiple steps and relationships are required to do these projections. In working backward from sample deltas, projected processor sample deltas are a function of the actual processor delay samples of an individual class and the actual wait-to-using ratio and projected wait-to-using ratio. The projected wait-to-using ratio of an individual class is a function of both the actual mean-time-to-wait of the class and the actual mean-time-to-wait of the class compared to the actual mean-time-to-wait of the

other work at the same dispatch priority. The projected wait-to-using ratio at a priority is a function of the processor demand of work running at higher dispatch priorities and the processor demand of work running at the same dispatch priority.

Actual delay samples, actual wait-to-using ratios, and actual mean-time-to-wait values are measurable. That leaves the problem of defining algorithms to project processor demand, wait-to-using ratios, and delay samples.

*Maximum processor demand*. The first problem with projecting the effects of dispatch priority changes is that the inherent processor demand of the work units in a class cannot be measured directly. If a class consumes *x* amount of processor service when it runs at dispatch priority *a*, it cannot be assumed that it will still consume the same amount of service when it runs at a higher or lower priority or with a more or less competing demand. The MGDPC required an algorithm to project the processor consumption of a class at any dispatch priority. The solution was to define the concept of maximum processor demand.

Maximum demand is defined as the theoretical maximum percentage of total processor time that work units in a class can consume if the demand has no processor delay. Its calculation follows:

maximum_demand_percentage

= number_of_work_units × processor_using_samples × 100 /
total_samples - processor_delay_samples

Maximum demand is calculated for each class and accumulated for all the classes at each priority.

*Wait-to-using ratio*. The next step in projecting processor sample deltas is to project the wait-to-using ratio that will be experienced by the classes at each priority given that one or more classes have tentatively changed priority. The aggregate projected wait-to-using ratio at a priority is a function of the processor demand of work running at higher dispatch priorities and the processor demand of work running at the same dispatch priority. The data used in the algorithm are the maximum demand of all the work running at each priority and the processor-using and delay samples accumulated by the classes at each priority. The current values of aggregate wait-to-using and aggregate maximum demand at each priority are used to determine the current functions relating wait-to-using to maximum demand. For each policy interval, these functions are derived dynamically to fit the current environment. Then the

dynamically derived functions are used to project the aggregate wait-to-using ratios expected to be experienced by the work at each priority after one or more classes and their demands are moved from one priority to another.

*Individual wait-to-using ratio*. Next the individual wait-to-using ratio for each class is calculated as shown below. The aggregate projected wait-to-using ratio at a priority was calculated above. The individual mean-time-to-wait was measured. Individual mean-time-to-wait is a function of the work units in the class and does not vary with priority. Service- weighted mean-time-to-wait is the sum of the products of individual mean-time-to-wait and individual processor service consumption with the sum divided by the total processor service consumption at the priority.

proj_ind_wait_to_using_ratio

= service_weighted_average_mean_time_to_wait /
individual_mean_time_to_wait

× proj_wait_to_using

*Processor delay sample delta*. Finally, projected processor delay sample deltas are calculated as shown below. The projected individual wait-to-using ratio was calculated above. The actual wait-to-using ratio was measured, and the actual processor delay sample value was measured.

proj_delay_samples

= proj_ind_wait_to_using_ratio /
actual_ind_wait_to_using_ratio

× actual_processor_delay_samples

The projected processor delay samples are equal to the actual observed processor delay samples, multiplied by the projected wait-to-using ratio, divided by the actual wait-to-using ratio. The delay sample delta is equal to the projected delay samples, minus the actual samples.

*Operation*. A state machine was developed to select and examine combinations of receivers and donors in order to identify and assess combinations of dispatching priority changes. The state machine is the mechanism used to determine whether the next priority move should be to move the receiver up, to move the donor down, to checkpoint interim changes, to commit final changes, or to select another donor. **Figure 5** shows an example of a state machine.

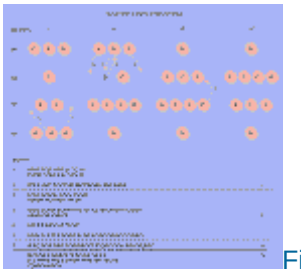


Figure 5

The initial donor is selected by the general select donor algorithm. Using that donor as a starting point, the processor fix algorithm alternately assesses the effect of increasing the dispatching priority of the receiver (moving the receiver up) and decreasing the dispatching priority of the donor (moving the donor down) until the combination of moves produces sufficient receiver value or insufficient net value. After each tentative priority change, net value is checked for all classes affected by the change. If all affected classes pass net value, the set of interim moves is checkpointed, and receiver value is checked. If there is insufficient receiver value, the state machine proceeds to select another tentative move for the receiver or donor. If the net value check fails after any tentative move, secondary donors and receivers are selected to be moved up with the receiver or down with the donor to determine whether that combination of moves will pass the net value check.

If at any point a priority change has a projected detrimental affect on another class, the affected class may become a secondary receiver and be moved up with the primary receiver. Multiple combinations of secondary receivers moved up with the primary receiver, and secondary donors moved down with the primary donor, will be considered to the extent necessary to find a combination of priority changes that will improve the receiver without causing relative harm to other workloads. The state machine handles all combinations of primary and secondary receivers and donors.

If moving secondary donors and receivers is still not sufficient to pass net value, the secondary donors and receivers are moved back to the most recently acceptable set of checkpointed priorities that had shown acceptable net value. Then if it was the primary receiver moving up that failed net value, the moves continue with the donor moving down. Conversely, if it was the primary donor moving down that failed net value, the moves continue with the receiver moving up. In both cases, secondary donors and receivers are selected after every move if required to pass net value and to allow the assessment to continue. If even with moving secondary receivers and donors, neither the priority of the receiver nor the priority of the donor can change with acceptable net value, the whole set of tentative and checkpointed moves is abandoned and another donor is selected by the select donor algorithm. Then the whole process starts over

with the new donor. The purpose of the state machine is to produce a comprehensive set of move combinations to evaluate, i.e., to leave no stone unturned in a search for changes to allow work to meet goals. However, in reality, the state machine tends to find valuable moves quickly because of the intelligence used by the select receiver and select donor algorithms when selecting initial candidates.

If a combination of priority changes with sufficient receiver value and net value is found, all the tentative priority changes are committed. The processor delay fix algorithm then exits and the MGDPC awaits feedback on the effect of its actions.

**Multiprogramming level delay fix.** This subsection describes how performance is improved by reducing the delay experienced by the receiver while it is waiting to be admitted to the multiprogramming set. An address space must be admitted to the multiprogramming set before it can be swapped in and execute. The controlled variable in this case is the number of MPL slots allocated to the class. One MPL slot represents one address space.

*Theory*. The MPL delay experienced by the receiver is a function of the fraction of ready users in the class that have MPL slots available to them. A ready user is an address space that is ready to execute. If there are fewer MPL slots allocated to the class than the class has ready users, some users will experience MPL delays. The class will not experience MPL delay if the number of MPL slots always equals or exceeds the number of ready users. The MPL delay fix algorithm uses an MPL delay plot to predict the effects on MPL delay of increasing or decreasing the MPL slots allocated to a class. At every policy interval, for each class, the MGDPC plots the most recent value of MPL delay per completion as a function of the fraction of ready users that have MPL slots available to them. **Figure 6** depicts an MPL delay plot.

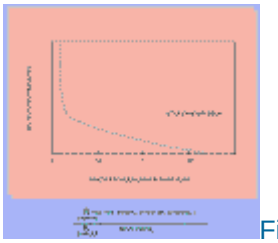


Figure 6

A complication arises when using the MPL delay plot because the number of ready users, required to read off the plot, is a function of the number of MPL slots. If there are too few slots, users will back up at any workload level. As slots are added, the number of ready users decreases. So the number of ready users is a function of MPL slots. The MPL delay fix algorithm uses another plot, the ready user average plot, to deal with this complication. The

ready user average plot **(Figure 7)** records the relationship between the number of ready users in a class and the number of MPL slots available to them. The ready user average plot is used to predict the number of ready users when assessing an MPL target change. The plot can show the point at which work units will start backing up. The number of ready users read off the ready user average plot is used to determine which point to read from the MPL delay plot.

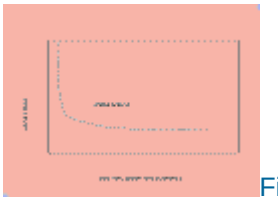


Figure 7

*Operation*. In the operation of this algorithm, first the MPL slot increase necessary to satisfy receiver value for the receiver class is found. This is done by adding one to the current MPL slot allocation of the class, using the new number of MPL slots to read the new number of ready users off the ready user average plot, using the new number of ready users to read projected MPL delay off the MPL delay plot, converting the new MPL delay to an MPL delay sample delta, using the new delay sample delta to project a performance index delta, and using the projected performance index delta to determine receiver value. If there is not sufficient receiver value, another MPL slot is tentatively added, and all the calculations are repeated.

When a number of MPL slots with sufficient receiver value is found, it is necessary to find storage to accommodate the additional swapped-in address spaces. Otherwise, simply adding address spaces could cause storage contention and other problems. Storage donors are identified using the find donor algorithm. The projected delay sample deltas are projected by the algorithm specific to the resource being taken (MPL slots or storage); the performance index delta is calculated as described previously; and the net value algorithm is used to determine whether the donation has value. If necessary, additional donors are identified and evaluated until an acceptable set of donors, able to donate the required storage, is found or all donors have been evaluated and all donations have been found to have insufficient net value.

If a change with sufficient receiver value and net value is found, the additional MPL slots are allocated to the receiver, and all the storage donations are committed. During the next policy interval, the receiver will use the new MPL slots, and the donors will be allowed to use less storage. The MPL delay fix algorithm then exits, and the MGDPC awaits feedback on the actual effect of its actions.

**Disk paging delay fix.** This subsection describes how performance is improved by reducing the disk paging delay experienced by the receiver. The controlled variable in this case is the number of processor storage frames protected for an address space. The protected number of frames is referred to as the protected processor storage target. The operating system will not steal frames from the address space below the protected processor storage target of the address space.

*Theory*. The disk paging delay experienced by the receiver is a function of both the number of page faults taken by work units in the class and the time required to satisfy the page faults. The number of page faults taken is a function of the processor storage allocated. The time per page fault is not a function of the processor storage allocated. It is a function of the demand put on the paging subsystem by all of the work units in any classes taking page faults. Both the number of page faults taken by the class and the time per page fault must be used in combination to accurately predict paging delay changes. The disk paging delay fix algorithm combines two techniques to predict disk paging delay changes. The first technique is the page fault rate plot, shown in **Figure 8**. This is a plot of page faults per completion as a function of processor storage allocated. A point is plotted on the class page fault rate plot after every few transactions in the class complete. The plot always reflects the latest condition but also remembers the page fault rate for the class when the class was allocated a larger or smaller number of frames. This plot is used to predict a new number of page faults per completion given a contemplated change to processor storage allocation.

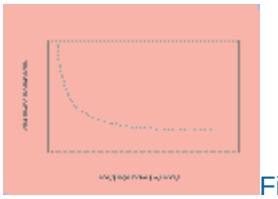


Figure 8

After a new page fault rate has been read off the plot, disk paging delay samples are used to predict the new time that will be experienced because of disk paging delay. This prediction is arrived at by taking the ratio of the change in page fault rate and multiplying it by the disk paging delay samples experienced by the class. The calculation follows:

projected_samples_delta

= (new_page_fault_rate - old_page_fault_rate)
× disk_delay_samples /
old_page_fault_rate

If a page fault is taking a long time because of other demands on the paging subsystem, this situation is reflected in the number of delay samples experienced by the class. Introducing disk samples into the algorithm serves the function of introducing time per page fault into the algorithm. The number of page faults and the time per page fault used in combination are accurate predictors of the disk paging delay that will be experienced by a class after a processor storage allocation change. The performance index deltas are calculated from the delay sample delta as described previously.

*Operation*. In the operation of the algorithm, first the storage allocation increase necessary to satisfy receiver value for the receiver class is found. This is done by reading the page fault rate corresponding to increasingly larger numbers of processor storage frames off the paging rate plot. Delay sample deltas and performance index deltas are calculated as described previously, and the receiver value algorithm is applied until a storage increase with sufficient receiver value is found. The required storage increase per address space multiplied by the number of swapped-in address spaces yields the total number of frames required. The storage required is found by the find donor algorithm, and the value of the storage reallocation is evaluated using the net value algorithm as described previously.

The paging rate plot captures the nonlinear relationship between the amount of storage allocated to work and the value of the storage to that work as measured by the page fault rate of the work. For work on the right side of this plot, additional storage will be of little benefit, whereas the same amount of storage could provide a very large benefit to work on the left side of the plot. All other things being equal, work on the left side of its paging rate plot will tend to be a receiver of processor storage, and work on the right side of its paging rate plot will tend to make a good donor of processor storage.

*Long-running transactions*. The preceding discussion assumed that each address space in the class would have similar storage requirements and benefit similarly from the same amount of storage. This assumption is true for classes where each transaction is relatively short. In these cases, each transaction in the class is given the same allocation of processor storage as soon as it arrives in the system. This technique allows short transactions to receive the benefit of the storage before the algorithms would have had time to learn the particular paging characteristics of each transaction. In cases where the transactions are longer, the transactions in a class, and

their storage requirements, are more likely to be different from one another. Also in this case, the algorithms can afford the time to learn about each transaction individually. To learn, the algorithms build a paging rate plot for each individual transaction that experiences significant paging delay. This plot is used similarly to the class paging rate plot. To project the effect of giving storage to a receiver, a new paging rate is read off the transaction paging rate plot. Then a projected number of samples is calculated for the transaction in the same way as a new number of paging delay samples was calculated for a class. The final step is to project the delay sample delta for the class by multiplying the delay sample delta for the transaction by the proportion of the paging delta of the class attributable to the transaction.

*Client/server considerations*. To improve the performance of a client class by reducing the paging delay seen by a server class, the delay sample delta is calculated as described above for the nonserved case. Then the projected samples are apportioned back to the client class, and a new proportional aggregate speed is calculated for the client class. The proportional aggregate speed plot is read to obtain the projected client class performance index and calculate the projected performance index delta as described previously.

*Feedback*. In all cases, if a change with sufficient receiver value and net value is found, the additional storage is allocated to the receiver by increasing the protected processor storage target of the receiver, and all the storage donations are committed. The storage donations may be in the form of reducing MPL slots or reducing protective processor storage targets. The disk paging delay fix algorithm then exits, and the MGDPC awaits the next policy interval to obtain feedback on the effect of its actions.

**Anticipatory resource allocation.** The previous topics on fixing storage-related delays all discussed how the MGDPC responded to situations where a class was experiencing delay waiting for MPL slots or paging. However, a good system programmer would not just wait for a problem with these resources to occur. The MGDPC does not just wait for problems either. The MGDPC anticipates what storage is needed by classes and allocates MPL slots and protective processor storage targets in advance to prevent problems. The MGDPC does these anticipatory allocations to the extent that the storage resource is not needed to solve problems that another class is experiencing. The MGDPC reconsiders these anticipatory allocations every policy interval, providing another mechanism for the MGDPC to respond to changing situations. The anticipatory allocations require no input from the customer. The MGDPC determines these allocations by observation.

**I/O delay fix.** This subsection describes how performance is improved by reducing the I/O delay experienced by the receiver. The controlled variable in this case is the I/O priority.

*Theory*. Managing access to I/O devices has many parallels with managing access to the processor. Both have using time and wait time, which suggested a wait-to-using algorithm. The concept of maximum demand, used successfully in processor management algorithms, is also applicable. This concept led to I/O priority assessment algorithms that in many ways paralleled the dispatching priority algorithms. I/O maximum demand and I/O wait-to-using measures are used, and the underlying concepts in the I/O projection algorithms are very similar to the concepts in the processor projection algorithms. A state machine is used to make a comprehensive search for I/O priority increase and decrease moves and secondary donors and receivers. The operation of this state machine is similar to the operation of the processor state machine described previously.

*Resources subsets*. There is a complication with I/O devices in that they, unlike processor and storage, are not a common resource. All work does not use all devices. If the MGDPC was going to affect performance by changing the I/O priorities of receivers with respect to donors, it had to know that the donor actually affected the receiver. If the receiver uses devices a, b, and c, and the donor uses devices x, y, and z, changing the I/O priority of the receiver with respect to the donor will have no effect. The MGDPC solved this problem by determining disjoint subsets (clusters) of I/O devices such that it knew, for example, that service classes a and b use the devices in cluster 1, and classes c, d, and e use the devices in cluster 2, and so on. The MGDPC dynamically builds these cluster and class relationships every ten minutes to reflect changes in how the work in the classes is using the devices.

*Multisystem shared resources*. Another problem involved with management of I/O priorities is that I/O devices, again unlike processor and storage, can be shared among systems. Managing I/O priorities on one system would be an incomplete solution in a sysplex. It led to the more general problem of being able to manage resources shared by multiple systems to meet performance goals.

When processor priorities are changed, the changes need only be done on one system because only work on one system is affected. However, when shared resources are involved, the changes must be propagated across all systems that share the resource. For example, if class a is running with an I/O priority of 253 on one system, it must run with an I/O priority of 253 on all systems to maintain its priority relative to other classes. If changes were not propagated across all the systems so work used consistent priorities, the changes on any one system would

have an unpredictable effect. The MGDPC solved this problem by coordinating the I/O priority changes.

A fundamental and very valuable attribute of the MGDPC algorithms is that the systems in the sysplex are independent as well as cooperating. There is no master system. Each system sees the same data and can make changes to any resource that the MGDPC manages. A complication arises with shared resources where any system can make I/O priority changes that it expects all systems to implement. The MGDPC solved this problem by continuing the philosophy that any system can make changes, but the MGDPC added coordination such that only one set of changes is propagated to all the systems at any one time. To reduce the instances of frequent competing I/O priority changes and to encourage the instances of the MGDPC to work on other problems such as storage or processor delays, the MGDPC added the requirement that the MGDPC on each system had to wait "*n*" number of intervals since the last I/O priority change by any system to make more I/O priority changes. The "*n*" used is six, so each system knows it has to wait six intervals before considering more changes. Maintaining the independence of the systems is very important because it allows each system to work on its local performance problems if all the sysplex goals are being met and eliminates many problems of a master-slave operation.

**Multisystem goal-driven performance controller in action.** This subsection describes an experiment that was run to show how WLM can manage a large commercial workload. The experiment had two phases. In the first phase, an on-line transaction processing and interactive workload was run. We discuss how WLM sets dispatch priorities for this work based on the goals and importance of the work. In the second phase, a large batch job stream was added to this mix. We next discuss how WLM adjusted to this change in the workload to continue to meet the goals of the WLM policy. It should be noted that in order to show the robustness of the WLM adjustment algorithms, the second phase of this experiment overloaded the processor capacity of the system in a way that a commercial environment with important on-line work would be unlikely to do.

The on-line transaction work consisted of two transaction-processing subsystems: Customer Information Control System Version 4.1 (CICS V4.1) and Information Management System Version 5.1 (IMS* V5.1). Both CICS and IMS are considered servers by WLM (see subsection on server topology). The interactive work was made up of 350 simulated users of the OS/390 Time Sharing Option (TSO). The batch work consisted of 10 large jobs designed to simulate commercial batch operations. This work was divided into two service classes, BatchHi and BatchLow. The workload was run on an IBM ES/9000*/9021 with two CPUs.

**Table 1** summarizes the WLM policy used for the experiment.

### Table 1 Goals for mixed workload with IMS more important

| Service Class Period | Type of Goal | Goal | Importance |
|---|---|---|---|
| CICSTRX | Response time | 0.090 sec | Medium |
| IMSTRX | Response time | 1.000 sec | High |
| TSO Period 1 | Response time | 0.100 sec | Medium |
| TSO Period 2 | Response time | 1.000 sec | Medium |
| TSO Period 3 | Response time | 3.000 sec | Low |
| BatchHi | Velocity | 7% | Lowest |
| BatchLow | Velocity | 1% | Lowest |

A significant observation about this policy is that the most important work in the system is made up of the IMS transactions. The least important work consists of the two batch service classes. The CICS transaction and TSO users are of medium importance.

There were two interesting phases to this experiment. First, the nonbatch workloads were started and stabilized. During this interval the system was about 80 percent utilized, and there were no storage constraints. The order of dispatching priorities that WLM chose for the work was:

1. CICS address spaces
2. TSO Period 1
3. TSO Period 2 and TSO Period 3
4. IMS address spaces

This order might be considered a surprising result given that the IMS transactions are the most important work in the system. The explanation is that although the IMS transactions are the most important, they are also very easily meeting their goal as shown by an average performance index of 0.12 over this interval. **Table 2** shows the average performance indexes of each service class and the percentage of the CPU that each service class was using during the first phase of the experiment.

### Table 2 Average performance index and CPU percentage

| Service Class |
|---|

| Period | Performance | |
|---|---|---|
| **Index** | **CPU (%)** | |
| CICSTRX | 0.70 | 24 |
| IMSTRX | 0.12 | 23 |
| TSO Period 1 | 0.52 | 11 |
| TSO Period 2 | 0.34 | 4 |
| TSO Period 3 | 0.31 | 8 |
| All | | work |

N/A

7

Notice that all the other service class periods have significantly higher performance indexes than the IMS transaction class (IMSTRX). If the IMS address spaces were given a higher dispatch priority, it would increase the difference between the performance indexes of the IMS transaction class and the other service class periods.

Now consider how a system programmer might go about setting the dispatch priority for this workload. Given that the IMS transactions are the most important work for this installation, the system programmer would probably not give the IMS address spaces the lowest dispatch priority. If IMS address spaces were given a higher dispatch priority, response time for at least TSO Periods 2 and 3 would be unnecessarily elongated, whereas the IMS transactions would beat their goal by even a larger amount.

If the system programmer did set the above dispatch priority order, the response time of the IMS transactions would have to be constantly monitored to look for a change in the workload that would cause the IMS transactions to miss their goals. If such a change did occur, the system programmer would have to detect it and decide how to change the priorities on the fly before too much damage was done to the IMS transactions.

The second part of the experiment was to start the batch work. With the batch work running, the overall processor demand of the total workload was significantly more than the system could deliver. There still was no significant storage contention. **Figure 9** shows how processor service was consumed by each of the different types of work during the overall run.

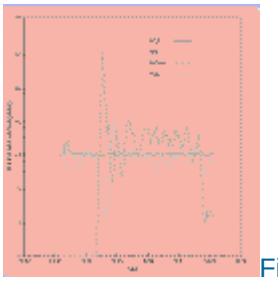


Figure 9

The batch work was started at about 15:47. Notice that the batch processor service immediately jumps to a peak of about 23000 with a corresponding drop in the processor service for the IMS address spaces and TSO work. **Figures 10**, **11**, and **12** show plots of how the performance index for the work changed during the run. Figure 10 is for IMS and BatchHi and Figure 11 is for the TSO periods. Figure 12 shows the same data on one plot. Note that the performance indexes for the IMS transaction class and the TSO service class periods shoot up as their corresponding processor service goes down. Because the IMS transaction class is the most important, WLM first addresses its problem by increasing the dispatch priority of the IMS address spaces relative to batch. The result of this action shows clearly in Figure 9 as the IMS processor service recovers as quickly as it dropped off. The processor service of TSO recovers after that of IMS since WLM addresses its processor delay problem as the next most important work missing its goal. After WLM went through several steps of incrementally improving the performance of TSO by adjusting TSO Period 2 and Period 3 dispatch priority versus the BatchHi class, the final dispatch priority order that WLM sets is:

1. CICS
2. TSO Period 1
3. IMS
4. TSO Period 2, TSO Period 3, and BatchHi
5. BatchLow

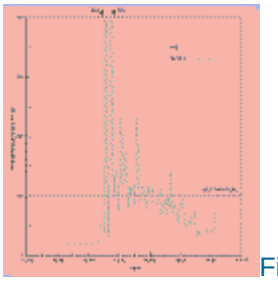


Figure 10

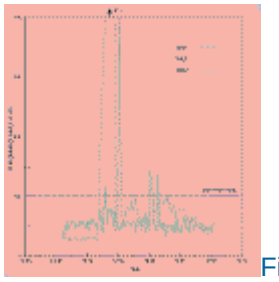


Figure 11

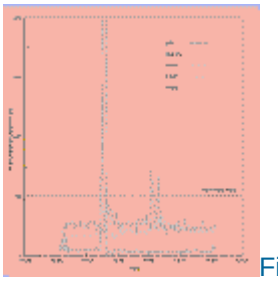


Figure 12

Examining the graph of performance indexes shows this dispatch priority order allows all the nonbatch work to meet its goals almost all of the time. **Table 3** shows the average performance

index and the percentage of the CPU each service class was using after the batch started and WLM had a chance to readjust dispatch priorities.

Table 3 Performance index and CPU percentage after batch started

| Service Class Period | Performance Index | CPU (%) |
|---|---|---|
| CICSTRX | 0.65 | 24 |
| IMSTRX | 0.07 | 22 |
| TSO Period 1 | 0.51 | 12 |
| TSO Period 2 | 0.45 | 4 |
| TSO Period 3 | 0.66 | 8 |
| BatchHi | 0.82 | 25 |
| BatchLow | 0.60 | 4 |
| All work | 0.60 | 9 |

Notice that the work that is most affected by the addition of batch operations is TSO Period 2 and Period 3. Before the batch work started, the average performance indexes for these periods were 0.34 and 0.31. After the workload has stabilized again following start of the batch, the average performance indexes for these periods increase to 0.45 and 0.66. Even the BatchHi service class is able to meet its goal on average, though it has the highest average performance index which is reasonable, since it is the least important work.

Given that overall this workload requires more processor power than is available, some work is not going to run. Since BatchLow is the least important work with the easiest goal, it is the work that is sacrificed. **Figure 13** shows the service rate of BatchHi versus BatchLow. Other than a small burst of service before WLM readjusted the priorities for the new work, BatchLow does not run until the jobs in BatchHi begin to finish at about 16:01.

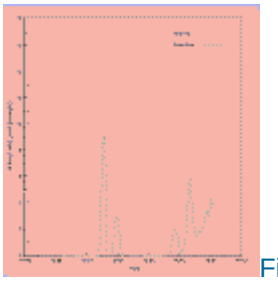


Figure 13

In summary, this example shows how the MGDPC function of WLM is able to allocate system resource to a diverse workload to meet the performance goals of the installation. Because the MGDPC is continually monitoring how the work in the system is performing, WLM can be more aggressive than a system programmer in reallocation of resources to the work in the system having the biggest problem meeting its goal even if it is not the most important. The order in which the MGDPC chooses receivers and donors and its net value check ensures that such a reallocation does not hurt more important work. The first phase of this experiment shows the results of such actions. The second phase of the experiment shows that WLM can quickly react to major workload changes in reallocated resources as necessary to best meet the performance goals of the installation.

### Balancing work across a parallel environment

A number of problems arise with the existence of multiple images that share work and resources. This section describes how WLM addresses these problems while remaining focused on the goals specified by the business policy.

In many interactive or client/server environments, a single user, while under some application suite, will remain "connected" to a particular instance of an application server running in the parallel environment for a protracted period of time. This time frame may extend for minutes or hours or days, depending on the nature of the application and the activities of the end user. Furthermore, the intended duration of this "connection" at the outset is unknown to the operating system or even the application itself in general. A further complication is that it is not known at the outset what will be the resource requirements of the end user, nor is it known what business goals and importance will be associated with this work.

Since workload conditions in the parallel environment may change while the "connection" exists, it is impossible to guarantee that a decision to "connect" the end user to a particular server instance on one image will remain optimal for the entire duration. For a variety of reasons, including network protocols, the existence of transient data, and recovery schemes, an end user

cannot be arbitrarily reconnected to another image or application server instance by the operating system in order to rebalance work when conditions change.

Prior to WLM, the techniques to solve this problem in an OS/390 environment involved planning the network connections so as to spread sessions across as many images as necessary to balance the workload. End users needed to be aware of which server or image they would be connected to in order to use their application. Some improvement was offered with Virtual Telecommunications Access Method (VTAM*) support of generic resource, wherein sessions (connections in a Systems Network Architecture, or SNA, world) were balanced across eligible logical units (an LU is an SNA session endpoint in this context). However, this support did not incorporate any knowledge of utilization or machine size. Techniques such as "round-robin" have a similar deficiency, and furthermore do not incorporate knowledge when "connections" are no longer active.

The approach taken by WLM in the face of the above limitations and uncertainties is to "connect" the end user to a server instance on an image that is meeting its goals and that has a threshold amount of "displaceable capacity" at the lowest importance level among eligible servers. Displaceable capacity at the lowest importance level refers to processor capacity that is either unused or that is consumed by recently observed work that is as low in business importance (as given by the business policy) as possible to achieve the threshold amount. The value for the threshold is dependent on the workload to which the user belongs and may be calculated based on samples that are taken as the workload runs, or may be based on historical values for the workload or on default values.

The principle behind this approach is that work requests created by the user will either have access to unused capacity or will compete with work that is deemed least significant by the installation business policy. In the latter case, the WLM algorithms will adjust resources, as needed, to ensure that the most important work achieves its business goals. This adjustment is the meaning of the phrase "giving work requests the 'best chance' to meet their goals."

Initially, this approach should give work requests created by the end user a maximal opportunity to achieve their goals. Over time, workload conditions may change and leave the end user's work less capable of meeting its goals, at least as compared to other server instances or other images. Of course, if these work requests are deemed to be sufficiently important by the business policy, the algorithms should ensure that sufficient resources are available to meet their goals. However, this condition leaves open the question of what to do for work requests

that are not sufficiently important when other images may now have unused capacity or less important work.

This problem is addressed using techniques that are workload-dependent. For example, some workload environments have a further layer of "transaction routing," wherein a work request arriving at an application server is then forwarded to another application server instance that is a better choice. CICS V4 and CICSPlex* System Manager (CP SM) cooperate to implement this approach as one example. Another approach is to dynamically change the "connection" based on the server's awareness of when this change can be done transparently. DB2* (DATABASE 2*) V4 is able to perform this change for distributed SQL (structured query language) requests to a remote data-sharing group, as it spreads such requests across those members in the proportions recommended by WLM. Indeed, DB2 will poll WLM on a regular basis to ensure that the distribution pattern matches current conditions. A third possibility is that the work request may be split up, or parallelized, to run on multiple processors (i.e., instruction streams) either within the same image or on different images. For example, DB2 V4 can parallelize queries in this fashion. As with "transaction routing" discussed below, some assessment must be made that using these techniques will overcome their cost.

Note that even when "transaction routing" can be implemented, it imposes an additional "hop" and therefore additional cost, which could reduce throughput and increase response time. It is therefore desirable to choose the target "connection endpoint" carefully, mindful that conditions will change, possibly unpredictably, and that elaborate analysis may be counterproductive.

WLM supports a number of environments that exhibit such "long-term" connections to a server through a variety of interfaces, including generic resource, domain name server, and sysplex routing.

The generic resource and domain name server allow a group of equivalent servers to be treated as a single entity by end users when requesting a "connection." Generic resource is used in the SNA world, where a "connection" equates to a session. During initialization, each server identifies itself as belonging to a particular group and gives the (LU) name(s) with which it is associated. Domain name server is used in the TCP/IP (Transmission Control Protocol/ Internet Protocol) world.

The OS/390 domain name system (DNS) implementation allows system administrators to set up a common host name for a set of OS/390 systems. When DNS is queried for IP address resolution, WLM services are used to choose the best system or server to place the new work.

In this way the DNS/WLM resolutions cause incoming TCP/IP requests to be distributed intelligently across the sysplex.

Sysplex routing allows a group of equivalent servers to be registered and monitored for purposes of routing individual work requests among its members. WLM will provide recommendations on what proportions should be allocated to each server within the group for a narrow window on the order of a few minutes in length. Users of this service are then able to spread individual work requests across multiple servers in these proportions so as to enhance their chance of meeting their goals.

WLM also provides interfaces to allow a product that coordinates and provides services for a collection of related server address spaces to query WLM for its recommendation on which server space is the best choice for a set of related work requests. Typically a daemon process within the product will interact with WLM to manage such work requests. This interaction moves the scheduling responsibility from the daemon to WLM, where work for the entire parallel environment is monitored and managed.

Each of these interfaces draws on common samples, measurements, and projections of system activity that are described next.

**Balancing data.** WLM implements its routing decisions and makes recommendations on the basis of five types of information. The first indicator is the presence of resource constraints in the recent past. These constraints would include shortages of processor storage, paging space (secondary storage in a UNIX** environment), etc., or dangerously high levels of paging or swapping. Systems with such problems are automatically given the lowest possible recommendation value to receive new work, since they will likely be unable to start new server instances and be otherwise unlikely to support an increased workload. In fact, such a system would be operating in a mode to shed work since it is seriously overloaded.

Next, WLM maintains a dynamic list of eligible servers, along with the image on which it is located, and any other information needed to uniquely identify each server instance. This list is updated not only with the startup and shutdown of servers themselves, but also with the unexpected failure of images within the parallel environment. This list is necessary so that WLM can recognize which servers exist and make a choice (or choices) among this list.

In addition to tracking the presence of servers, WLM evaluates the ability of each server to meet the goals of work requests that have flowed through the server using an aggregated performance index (PI). This PI takes into account the various importance levels for such work

requests and their contribution to the universe of work requests that the server accepted in the recent past, and projects what the PI will be as a result of the policy actions taken. The significance of the PI is that it incorporates the effect of all activity in the system and reflects the real delivered responsiveness measured against the business goals. Delays include contention and constraints for all resources, including storage, locks, queuing effects, etc.

The third type of information used by WLM is the importance level service summary table, which tracks the normalized processor consumption of work at each importance level on each image. To be more precise, the cumulative processor service delivered to all work at the given level and for all less important work is maintained. This table includes unused capacity, discretionary work, and system overhead not directly attributable to any particular work request. The purpose of this table (see **Figure 14**) is to allow the WLM algorithms to understand where "displaceable capacity" exists on an image-by-image basis, and where work may compete most favorably for processor access.


Figure 14

The fourth type of information is an assessment of the average cost of each work request that flows through the server. For some work environments, this assessment may be based on historical measurements or, alternatively, may represent a default cost associated with a single end user. For other environments, this assessment may reflect the measured average cost of an end user over some recent interval. The purpose of this estimate is to set the threshold value for "displaceable capacity" needed, and also to estimate the latent demand that arises when a new "connection" has been established but before actual demand shows up in new measurements kept in the importance level service summary table and the aggregate PIs.

The fifth type of information used by WLM is the list of recent selections that allows some projection of latent demand of new connections, as discussed in the previous paragraph. This information is aged out fairly quickly as new measurements pick up the actual demand that has been introduced.

The above information is maintained on an ongoing basis during normal system execution. When a WLM interface is invoked to make a recommendation, as described above, WLM will also calculate the target service value for "displaceable capacity" that is needed. This value incorporates both the average cost and an estimate for latent demand that is reflected by the list of recent connections.

**Balancing algorithm.** With the above data in hand, the general approach in deciding how to place a new "connection" is to go through the list of all servers one at a time and assess whether the current server is a better choice than a server previously chosen. A server on an image that is not resource-constrained will be given preference to a server on a resource-constrained image. Beyond that filter, preference is given to servers that are meeting goals, then to servers that are narrowly missing goals, and finally to servers that are badly missing goals. Within each of these three categories, servers are ranked according to the importance level at which the target "displaceable capacity" is achieved, with preference given to utilizing the lowest-possible business importance.

If the interface allows multiple selections (as for sysplex routing), the algorithm will keep all selections that survive with the same attributes for resource (un)constraint, degree of meeting goals, and target importance level. Weights are set according to the ratio: (target displaceable capacity)/(total capacity of all images at target importance level).

The rationale for the above ordering reflects a number of trade-offs. As has been discussed previously, systems that are recently resource-constrained are immediately shunned since they are overloaded, and the system is actively reducing the workload that is allowed to run by disallowing new spaces from being created, reducing the number of spaces that are swapped in, and so forth.

In the absence of a resource constraint (or when all relevant images are similarly constrained), the ideal server choice is one that is meeting goals for its work and that resides on an image with sufficient displaceable processor capacity to accommodate new work. Ideally this would be unused processor capacity, but in any case, there is a preference to compete with work at the lowest possible importance level so that new work will be favored as much as possible.

A server that is narrowly missing its goals but with sufficient unused processor capacity to accommodate new work is almost as good as a server that is meeting its goals, since the resource management algorithms will likely address the problem--which would generally be one of processor storage allocation.

In looking at the above categories used in ranking server choices, it is worthwhile to observe that a strong reliance is placed on the actual performance of servers against the goals of the work they serve, as measured and projected by their aggregate PI. This observation reflects a philosophical bias to use actual observed behavior and to value feedback so as to correct inaccurate assumptions that might be made from other measurements or design points. This

concern has been discussed in other papers. Reference 9 discusses sensitivity to inaccuracy in the values of communication costs, locality statistics, etc. Also observe that the number of separate factors in making a decision is a mere handful. This latter observation reflects a second philosophical bias mirrored in Reference 7 relative to overly complex algorithms.

**Balancing algorithms in action.** We now describe an experiment designed to show how WLM allocates work across clustered systems. The experiment was run on three IBM 9672 Parallel Enterprise Servers. Each 9672 had six CPUs. At the start of the experiment, CPU-intensive jobs were started on two of the systems. Next, 1000 simulated users were logged on to the parallel system. We discuss how WLM distributes these new users to achieve workload balancing across the three systems.

In **Figure 15**, Systems A, B, and C are receiving work to establish OS/390 userid log-on sessions. System A has much more idle capacity than System B or System C. The CPU-intensive jobs running on B and C are absorbing the capacities of B and C. **Figure 16** shows that as new userids request to log on, WLM places them on System A until the idle capacity of System A falls below that of Systems B and C. WLM then places new log-on sessions on System B because at time 8:32, System B has more idle capacity than A or C. At time 8:38, the new users on System B have reduced the idle capacity of B, and the graph in Figure 15 shows an amount of idle capacity equal to System C. WLM now directs new log-on sessions to System C.

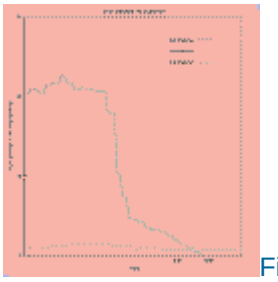


Figure 15

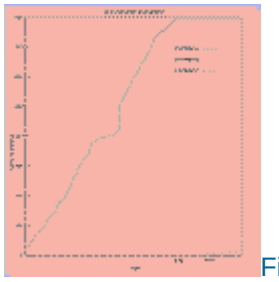


Figure 16

This experiment shows the effect of capacity considerations on the balancing algorithms of WLM and the responsiveness of the algorithms. Within microseconds, new log-on sessions are sent to the systems with better capacity. A human operator could not be as responsive or vigilant 24 hours a day, every day.

**Cooperating products on WLM environment.** MVS/ESA SP5.1.0 provides the initial support to allow work managers and resource managers to cooperate with WLM to surface: completions of business units of work, the associated delays as viewed by the subsystem product, and

which address spaces are involved in processing each work request. The following products utilize these new interfaces to make this possible:

1. CICS V4.1
2. CP/SM V1.0
3. IMS V5.1 (transaction manager and database)
4. Resource Measurement Facility (RMF* V5.1)

The following traditional work environments are also supported for goal definition and management in MVS/ESA SP5.1.0:

1. APPC/MVS scheduler (ASCH)--Advanced Program-to-Program Communication initiators for client/server environments
2. Job Entry Subsystem (JES)--batch work
3. OpenEdition* MVS (OMVS)--OpenEdition work utilizing UNIX and other programming semantics
4. Started Task Control (STC)--started tasks
5. TSO--interactive environment

MVS/ESA SP5.2.0 provides the ability to balance sessions via generic resources or balance work requests via sysplex routing. It also supports splitting of work requests either on a single OS/390 image or on multiple images. Other support allows products to communicate the presence of user requests to WLM for individual management within a single server as in the following:

1. CICS V4.1--generic resource
2. DB2 V4.1--distributed DB2, single-CEC (central electronics complex) parallelism, sysplex routing, generic resource
3. DB2 V4.2--multi-CEC parallelism, TCP/IP routing
4. VTAM V4.2, V4.3--generic resource
5. RMF V5.2

OS/390 R3 provides the ability to dynamically control the number of server address spaces on an application environment level based on goal satisfaction of the work requests queuing work to these servers. Related services allow products to dynamically route work requests to the optimal server address space. Additional support was added to allow server address spaces to separately identify multiple business units of work for classification and WLM goal management. The following products utilize these and related services:

1. Local Area Network File Server (LFS)
2. System Object Model (SOM*)
3. Internet Connection Secure Server (ICSS)
4. DB2 V4.2--WLM-managed stored procedures
5. RMF V5.3
6. TSO generic resource

## Conclusion

The systems management burden upon installations supporting large-scale enterprise-wide computing is growing at an alarming rate. The cost of configuring system hardware and software, application programs, and network configurations escalates with the introduction of new servers, new networking technologies, and new application program development technologies. The ability to absorb new technologies is vital to those wishing to exploit the possibilities offered--to gain business advantage. This paper described a technology to assist such exploitation. In addition, three significant dimensions are worthy of discussion: a strong supporting philosophy, an initial product implementation, and a dream of what the future may hold.

The foundation for workload management capabilities described in this paper is a crisp definition--to operating system software--of the underlying business rationale for employing a computer system. Knowledge of goals for work and the business importance of that work provides a well-informed basis for operating system software to take direct actions having positive business value to the purchaser of that system. The underlying philosophy is that the system should manage itself, using all available means, toward those business goals, with no additional requirement for human intervention. This workload management philosophy is in sharp contrast with other efforts within the industry, efforts focused on delivery of tools to aid information technology professionals with the optimization of critical resources. Low-level, detailed focus on individual application servers and the resources consumed by the servers enables one to quickly lose sight of what is really important: satisfaction of business needs.

Much information can be found in the literature describing theoretical problems and solutions, evaluations, and comparisons of alternative system structures, including descriptions of areas for valuable future research. The content of this paper has been limited to subjects embodied in software product implementations being used in commercial data processing environments. The functional capabilities required by those users are much greater than the topics reviewed in this

paper; detailed descriptions of many capabilities were omitted to limit the length of this paper. Some of the additional functions are:

- The ability to dynamically instantiate application server processes based upon trade-offs between demand, the ability to satisfy goals for the work requests that the servers serve, and the availability of system resources. This function eliminates the need for preconfigured application servers while preventing potential problems caused by excessive or inadequate application serving capabilities.
- The ability to manage a given work request as a single unit of work, even though the work request requires the services of more than one application server. This function allows the system to manage more closely to the requesting client view of a response time and eliminates the need to set individual performance goals for each application server.
- The ability to set an overall resource consumption limit for a set of service classes. This function provides a mechanism to guarantee availability of an amount of capacity for one or more critical workloads.
- The ability to temporarily promote the "importance" of an individual work request when that request has acquired a serially reusable resource needed by more important work. This function aids nondisruptive management of resource contention.
- The ability to generate detailed descriptive information showing resource consumption information and resource delay information in the context of the various service class goals in effect. This function explains why the actual results were achieved and provides a basis for capacity planning and system performance modeling tools.

Beyond the capabilities of the current System/390 workload management implementation lie future opportunities for expanding the quality of heuristic decision-making and the scope of resources being controlled. These opportunities represent much more than mere enhancements to the OS/390 operating system--they represent the implementation of a dream where information technology resources adapt themselves to satisfy business objectives without requiring human guidance beyond definition of those business objectives. As long as the need exists for detailed, low-level performance control parameters, the possibility of incorrect or inappropriate definitions will exist. Is the realization of such a dream possible? Maybe. It is easy to envision numerous functional extensions that fit neatly into the framework of the existing OS/390 workload management implementation. Some of these extensions are:

- To dynamically manage the size of memory-resident buffer pools, balancing the availability of memory against the value received from avoiding physical I/O activity and the cost of managing the buffer pools
- To more closely coordinate the scheduling of interrelated "networks" of jobs, managing an entire set of jobs toward a specific time-of-day completion requirement
- To retain longer-term histories of resource utilization patterns, so that repeatable peaks and valleys in workload demands can be anticipated

Although these problems appear to be solvable, other more complex problems remain. Complete goal-oriented management of an enterprise would require end-to-end management of distributed, heterogeneous operating systems and the network interconnection mechanisms that join these systems. Although the infrastructure exists within OS/390, varying amounts of capabilities exist on other operating system platforms. An end-to-end perspective implies the inclusion of:

- End-user workstations having little or no programming capabilities
- Network gateways and intermediate servers that currently have no understanding of the "work" that they process
- Operating system platforms having primitive resource management controls

Focusing on the operating system platforms alone does not address the full set of requirements, since the interconnection mechanisms must also manage the available bandwidth toward the needs of the work requests associated with data being transported.

These longer-term desires represent significant technical challenges.

**A New Computer Program Classifies Documents Automatically**

---

As the amount of documents and other data that companies amass continues to grow, employees searching for useful information on internal corporate networks, or intranets, increasingly encounter the same problems facing users of the Internet. Until recently, the primary approaches to dealing with this chaos have been either cataloging by trained librarians, which can be expensive, or semiautomated sorting based on preselected categories, which is less accurate and therefore less useful.

Now, however, two computer scientists at IBM's Almaden Research Center have developed a third way— a computer program that can analyze thousands or even millions of documents and create a taxonomy for the entire collection, with categories, subcategories, sub-subcategories, and so on — that allows users to quickly zero in on documents of interest. The brainchild of Shivakumar Vaithyanathan and Byron Dom, the program, code-named Sabio from the Spanish word for "wise one," actually considers many categorizations and settles on one that is highly efficient in terms of how easily users can find relevant documents. Sabio is being incorporated into a knowledge-management program, code-named Raven, under development by Lotus® that will provide a suite of tools for organizing, searching and viewing information in a company's intranet.

"The inspiration for Sabio came from the Internet search engine Yahoo!®" says Dave Newbold, general manager at Iris Associates, a Lotus subsidiary that develops Dominoª and Lotus Notes®. Yahoo! categorizes Internet sites, and users can search either by working through the organization tree to find the relevant categories or by performing a keyword search, identifying particular sites, and then looking at other sites that fall in the same categories. "A lot of corporations want to do what Yahoo! does, but Yahoo! was created manually, an option that is not feasible for most firms," Newbold says.

Thus, two years ago, during the development of Raven, Lotus held a competition for software that could organize the contents of an intranet into sensible categories, and

create a catalog of the contents based on those categories. Lotus extended invitations to a dozen vendors, including teams from IBM, to submit such programs, and eight accepted the challenge. They were then compared on how fast and how accurately they could categorize a set of 72,000 newswire releases on a variety of subjects with nothing to go on but the contents of each release.

The entries relied on a variety of methods. Vaithyanathan and Dom's program employs Bayesian statistics, a powerful technique that allows one to take into account educated guesses about what a solution will be and then modify those guesses according to data.

Sabio's first step, Vaithyanathan explains, is to decompose each document into a collection of "tokens," its relevant words and phrases, leaving out such inconsequential components as "and," "the" and "on the other hand" and assembling a collection of all the thousands of relevant words and phrases in all the documents. Sabio treats this collection mathematically as points in a huge multidimensional space, in which each dimension corresponds to a single word or phrase, and the number of times the word or phrase appears determines how far out along the dimension the point lies.

Only when two documents share many of the same words and phrases will they be relatively close together in this multidimensional space. The program therefore regards clusters of nearby points as being about similar subjects. In this respect, Sabio is similar to a number of other categorizing programs, Dom notes, but there is a crucial difference. "With most clustering programs, you have to indicate how many clusters to use and what features (i.e., which words or phrases) to cluster on." By contrast, Sabio figures out not just which documents are assigned to which clusters, but the number of clusters and how each cluster is defined in terms of key words and phrases. Some words and phrases, for instance, will probably turn up with similar frequency in almost all the documents and be of little use in clustering; others will occur regularly in only a certain subset of the documents and be crucial in picking out clusters of related documents; and still others will have a more complicated pattern of occu rrence. Sabio determines for each set of documents which words and phrases are useful and how they are used to define the clusters.

This flexibility allows Sabio to analyze a set of documents and, unaided by human categorizers, come up with an effective way to group the documents into relevant categories, but it also introduces a tricky complication. If the cluster model — the number of clusters and the collection of key words and phrases — is predefined, it is a

relatively simple exercise in statistics to find the best clustering. One simply minimizes the total "scatter," that is, how spread out the documents are inside the clusters in the multidimensional space of words and phrases. But if the cluster model is not fixed, the search for the best clustering must also take into account the complexity of the rules specifying which documents are put into which clusters. This leads to a delicate balancing act.

At one extreme, the scatter can be minimized by having almost as many clusters as documents, with each narrowly defined cluster containing only one or a few very similar documents. Such a complex clustering would be no improvement over the original set of documents, however, since it would not help users to zero in on items of interest. At the other extreme, the simplest possible clustering is to lump all documents into a single large cluster with a very large scatter. It, too, would be of no practical use.

What Sabio does, Dom explains, is to use Bayesian statistical techniques to find the best clustering of documents. In this case, "best" is defined in terms of a balance between keeping the scatter low and the clustering model simple. This is a new approach to clustering, Vaithyanathan says, and it has proved superior to any automated approaches. In tests on document collections that have been categorized by human experts, Sabio's clustering has come quite close to that of the experts, he says, and, as the number of documents increases, Sabio's performance improves.

Once the best way to cluster the documents has been found, Sabio creates a taxonomy by grouping related clusters, again using a balance between scatter and the complexity of the clustering rules to decide which clusters are most closely related. The result is a treelike structure in which a company's documents — the trunk of the tree — are split into perhaps a dozen major categories, or limbs, each of which is divided into branches, subbranches and so on, to the smallest twigs of the tree, which are the individual clusters of documents. To zero in on particular documents, a user simply moves up the tree, following the branches of interest.

All of this is done automatically, but Sabio also allows people to have input into the process in a number of ways. Once the categorization is complete, it is expected that users will review the categories and modify them to better suit themselves; for example,

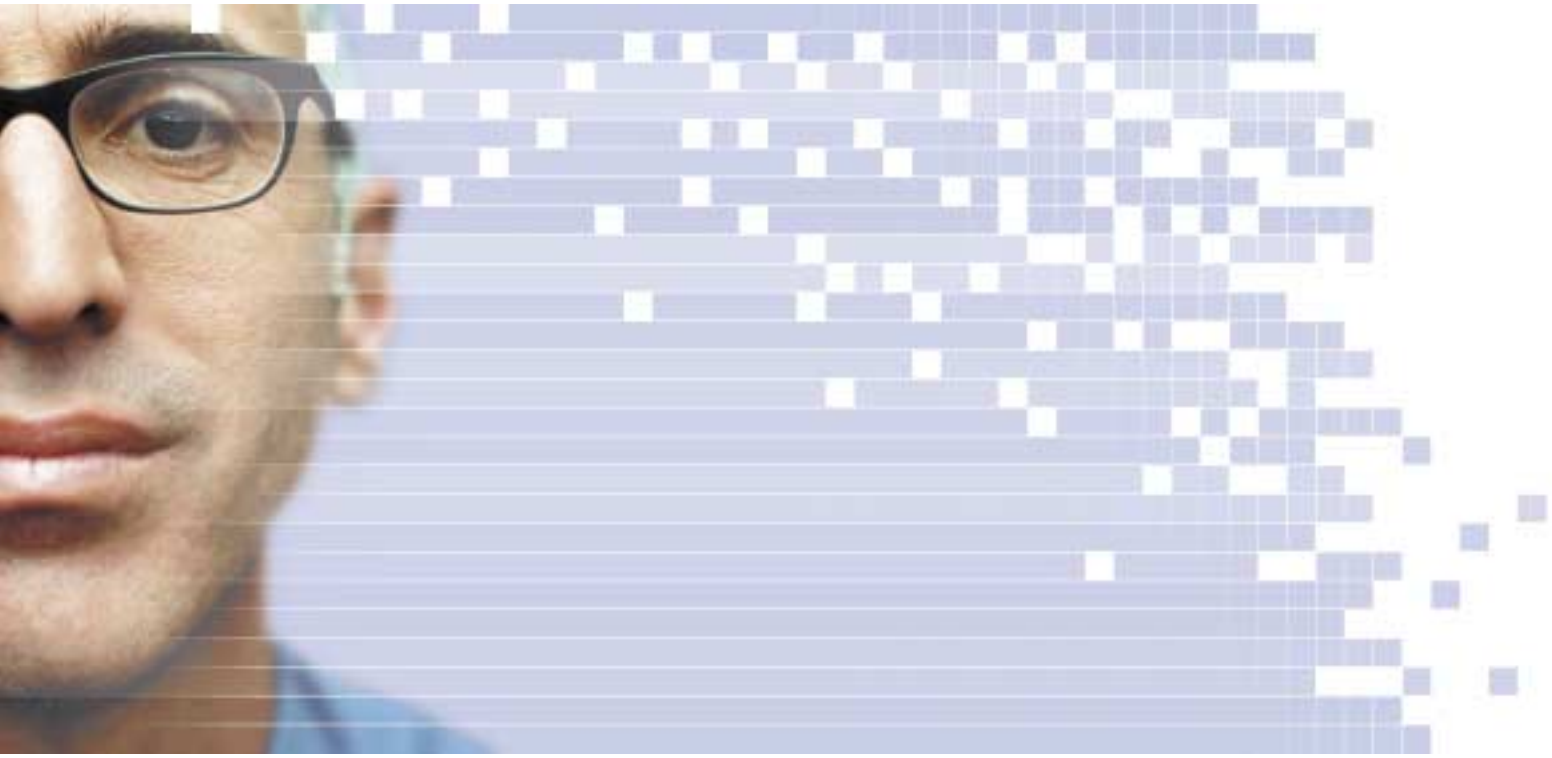
by combining categories or creating new ones or by moving certain documents from one classification to another. In each case, Sabio is programmed to learn from these modifications and to classify future documents accordingly. A user can also show Sabio a set of documents that has been categorized by human experts, and the program will generate a set of rules to produce that categorization and will apply those rules to other documents.

Sabio proved its mettle in the Lotus competition. Most important for Newbold and Lotus, Sabio's classification proved to be both fast and accurate. "It didn't take long to figure out who won the competition," Newbold recalls. "Sabio won hands down, in terms of both speed and accuracy. It was 80 percent faster than its nearest competitor and, although there is not a good numerical measure of accuracy in clustering, Sabio was clearly far better than the others at putting documents into the right categories."

The number of documents that Sabio can categorize is limited only by a company's computing power, Vaithyanathan points out. The accuracy actually improves as the number of documents increases, but very large sets of documents, such as those found in a company the size of IBM, demand a correspondingly powerful computer to handle them.

Sabio, or the Automated Taxonomy Generator (ATG) as it is called in Raven, will be combined with several other features in the Lotus product. Besides ATG, Raven has a full-text search engine that pulls up any document with a given word or words, as well as an expertise locator that uses information in the documents to identify people with knowledge of, or interest in, particular subjects. But the feature most likely to catch the attention of chief information officers is Sabio's ability to bring order to the most chaotic set of corporate e-documents.

IBM Server Group



# eLiza: Building an intelligent infrastructure for e-business.

*Technology for a self-managing server environment*

business

# Yesterday's fantasies are today's necessities

*Your information technology (IT) environment is getting more complex. Bigger and larger systems, billions of users and colossal transaction volumes are creating additional hard-to-fit pieces for the already complicated IT puzzle. Add to that workloads that are increasingly data-intensive, and it's clear that your infrastructure needs to do more than process data faster—it also needs to be smarter.*

*These conditions are accelerating the idea of self-managing, electronic "intelligence" from the subject of science-fiction novels to a real-world business priority. The IBM eLiza project is proof. An IBM @server initiative, eLiza strives to make your e-business infrastructure an autonomous, self-managing system. eLiza extends the IBM @server platform to help:*

- *Provide resource management and automation capabilities*
- *Configure your system on the fly*
- *Repair problems online*
- *Defend against unauthorized access*

*Innovative eLiza technology is embedded across IBM @server xSeries™, IBM @server iSeries™, IBM @server pSeries™ and IBM @server zSeries™ servers.*

## eLiza is more than just a vision for the future—our customers are already seeing the results.

*"We had a number of issues we were* ***dealing*** *with: multiple clients, multiple processes and the necessity for 24x7 availability. We found a solution for all of those* ***issues*** *in IBM @server iSeries and eLiza technology."*

*—David M. Howard, President and CEO, IMSG*

**Well-being in a self-help box**

Insurance Management Solutions Group (IMSG) offers business process and e-commerce outsourcing to keep many of the largest U.S. insurance carriers on the leading edge of their industry. To stay ahead of its own competitors, the company realized that it needed to rapidly Web-enable its quoting, policy management and reporting applications. IMSG also needed to ensure these applications would be available virtually 24x7.

IBM provided an end-to-end solution, including iSeries servers, IBM WebSphere® Application Server and IBM DB2® Universal Database™. Self-optimizing technology included in the IBM eLiza initiative helps ensure that this solution meets IMSG's requirements for nearly continuous uptime. iSeries logical partitions (LPARs) will spread the company's applications and dynamically allocate resources throughout the system to achieve high levels of availability and redundancy.

*The self-optimizing eLiza technology used by IMSG includes:*

- Logical partitioning—combines with self-healing clustering technology to remove the physical limits of a single server.

*Additional self-optimizing eLiza technology:*

- Self-managing performance—offers the choice of manual or completely automated server monitoring and management services.
- Intelligent Resource Director (IRD)—allows dynamic resource allocation on zSeries servers.

**Brainy building blocks**

When Sea Island needed to implement new Microsoft® Windows®-based applications to enhance its customer service capabilities, the world-class residential and resort destination turned to a proven partner for help. IBM established a robust, integrated environment of iSeries and xSeries servers and Integrated xSeries Adapters (IXAs) that make it easy for staff to manage the growing IT system.

The self-configuring characteristics of the underlying eLiza technology also help preserve the resort's investment in its existing systems. IBM IXAs extend automated systems management within the iSeries to the xSeries environment, allowing the resort to implement Windows-based applications while capitalizing on the storage, system and user-management capabilities of the iSeries. Backup and restarting is also done without human intervention.

*The self-configuring eLiza technology used by Sea Island includes:*

- Integrated xSeries Adapters for iSeries—includes hot spare support, dynamic storage management and high performance backup and recovery.

*Additional self-configuring eLiza technology:*

- Plug and Play—allows for the dynamic addition or deletion of processors, storage and input/output (I/O) devices.
- Capacity Upgrade on Demand—provides instant access to additional processors or servers.
- Setup Wizards—enable self-installation capabilities in servers and software.
- Wireless Systems Management—makes possible "anytime, anyplace" server management via wireless devices.

**An automatic spoonful of sugar**

Memphis City Schools, the twentieth-largest metropolitan school system in the nation, used IBM xSeries servers to consolidate approximately 180 servers into a single location. The new environment is easy to run and helps the district provide its users with the best possible educational experience.

The IBM @server environment also features self-healing eLiza technology to help the district's IT team avoid system failure. These powerful tools, including IBM Director and Light Path Diagnostics™, also help the district reach its goal of 100-percent uptime by streamlining and automating management and support tasks. IT personnel are now able to troubleshoot systems before they become critical.

*The self-healing eLiza technology used by Memphis City Schools includes:*

- Chipkill™ memory—recognizes memory failures and helps provide automatic problem detection, disablement and problem resolution without any server downtime.
- Clustering—enables high availability and enhanced scalability. Blue Hammer and Parallel Sysplex are features supported today on pSeries and zSeries. Opticonnect is featured in iSeries and clustering services are available for xSeries.

*Additional self-healing eLiza technology:*

- Call home—provides the ability to request service before or after a failure without human intervention.
- ECC Cache—detects and corrects errors within the storage hierarchy.
- Multipath I/O—gives your server(s) a variety of routes to reach information stored within the system and maintain overall system availability.

*"A part of 'intelligence' is the ability to* ***interact*** *in a structured manner with your surroundings. eLiza and IBM provide the global framework that allows us to securely and* ***efficiently*** *integrate, manage and run distributed or even heterogeneous systems."*

*—Claus Jensen, Senior IT Architect, Danske Bank*

**A virtual black belt in self defense**

Serving global banking customers through widespread operations, Danske Bank recognized that a secure, efficient and available IT environment goes a long way in retaining existing customers and attracting new ones.

An IBM @server environment with zSeries and pSeries servers helped Danske Bank establish an effective "one user, one identity" security strategy. Easing the verification and authentication workload for Danske Bank's IT personnel is self-protecting eLiza technology, including Secure Sockets Layer (SSL), Remote Access Control Facility (RACF) and digital signature. Danske Bank employs a home-grown single signon solution leveraging the zSeries coupling facility, and plans to adopt IBM single signon and Lightweight Directory Access Protocol (LDAP) tools in the near future.

*The self-protecting eLiza technology used by Danske Bank includes:*

- SSL—manages Internet transmission security.
- Digital Certificates—provide identity authentication.
- Encryption—helps to prevent unauthorized use of data.
- RACF—limits remote system access to authorized users.

*Additional self-protecting eLiza technology:*

- Single signon—permits access to multiple applications with one user name and password.
- LDAP—aids in the location of network resources.
- Kerberos—authenticates requests for service in a network.

**IBM and eLiza help you get the most out of your IT**

Admit it, you talk to computers. It's O.K., everybody does. Most of these conversations, unfortunately, are not only one-sided but probably happen under unhappy circumstances. IBM technology, however, is helping to turn this around. Across the globe, businesses are making their infrastructure smarter and easier to manage by building on their existing IT platforms with IBM @server systems and eLiza technology. In the complex and constantly evolving world of e-business, information is your most valuable asset. IBM can help you get the most out of it.

**For more information**

To learn more about the IBM @server platform, visit:

**ibm.com**/eserver/

To learn more about the eLiza initiative, visit:

**ibm.com**/servers/eliza



IBM Server Group
3039 Cornwallis Road
Research Triangle Park, NC 27709

Printed in the United States of America
10-01


References in this publication to IBM products and services do not imply that IBM intends to make them available in all countries in which IBM operates.

These customer stories are based on information provided by Danske Bank, Insurance Management Solutions Group, Memphis City Schools and Sea Island, and illustrate how one organization uses IBM products. Many factors may have contributed to the results and benefits described; IBM does not guarantee comparable results elsewhere.

IBM @server systems are assembled in Australia, Brazil, Great Britain, Japan and the U.S. and comprise U.S. and non-U.S. components.



G520-9592-00

# Gryphon:

## Publish/subscribe over public networks

**IBM T. J. Watson Research Center**

For additional information please contact:
gryphon@watson.ibm.com

Gryphon (pronounced \'gri-fin\) is publish/subscribe middleware aimed at distributing large volumes of data in real-time to thousands of clients distributed throughout a large *public network*. A public network is a wide-area extranet or intranet that is too large or complex to be centrally administered to support specific applications. Such a network contrasts with well-controlled, centrally administered networks that have traditionally been the deployment environment of publish/subscribe systems such as TIBCO's Rendezvous.

Gryphon has been deployed over the Internet for real-time sports score distribution at the US Tennis Open, Ryder Cup, and Australian Open, and for monitoring and statistics reporting at the Sydney Olympics.

### Technical Differentiators

There are four areas of focus for Gryphon that distinguishes it from traditional publish/subscribe systems:

- Gryphon provides both topic-based and content-based publish/subscribe. Content-based subscription is the capability of a client to request messages based on their content. Topic-based addressing is an abstraction of numeric network addressing schemes. However, as applications evolve (Figure 1) and as new application are added, the addressing scheme must also change, requiring existing applications to be modified. With content-based subscription (Figure 2), delivery depends only on the content of messages, and therefore, applications can select different combinations of messages without changing an addressing structure.

- A publish/subscribe system deployed on a public network cannot depend on homogeneous router technology. In particular, reliance on LAN or IP multicast technologies is impractical in a large public network. Instead, universally adopted standards such as TCP/IP or HTTP must be used for all communication.
- A publish/subscribe system should be able to scale to support application growth. This includes supporting a large number of clients connecting to a single site and supporting linking of geographically distributed sites. As systems scale, the probability that a component will fail increases and a scalable publish/subscribe system must be able to recover from these failures.
- A publish/subscribe system deployed over a public network must provide security and privacy features to a degree not mandated over private secured networks. These features must include client authentication, access controls, and encryption/integrity of messages.

In Gryphon, functionality for these requirements is combined with a flexible information model supporting very large application deployment and application evolution. Below, we describe the features of Gryphon.

**FIGURE 1.**

Topic-based publish/subscribe systems limit application evolution. In this case, a subscriber interested in high-volume stock trades must subscribe to all topics and filter thousands of events per second at the client.

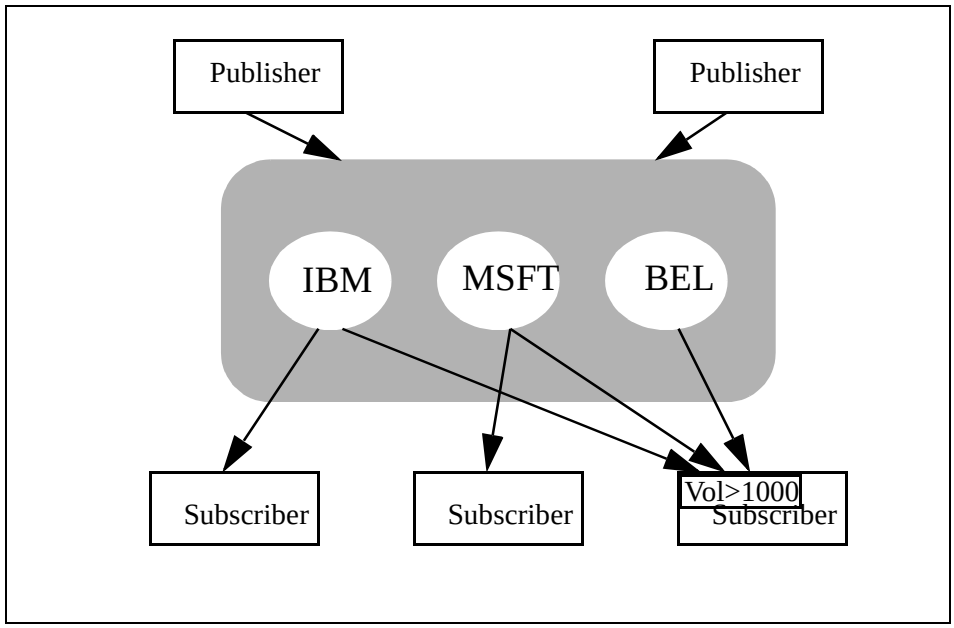
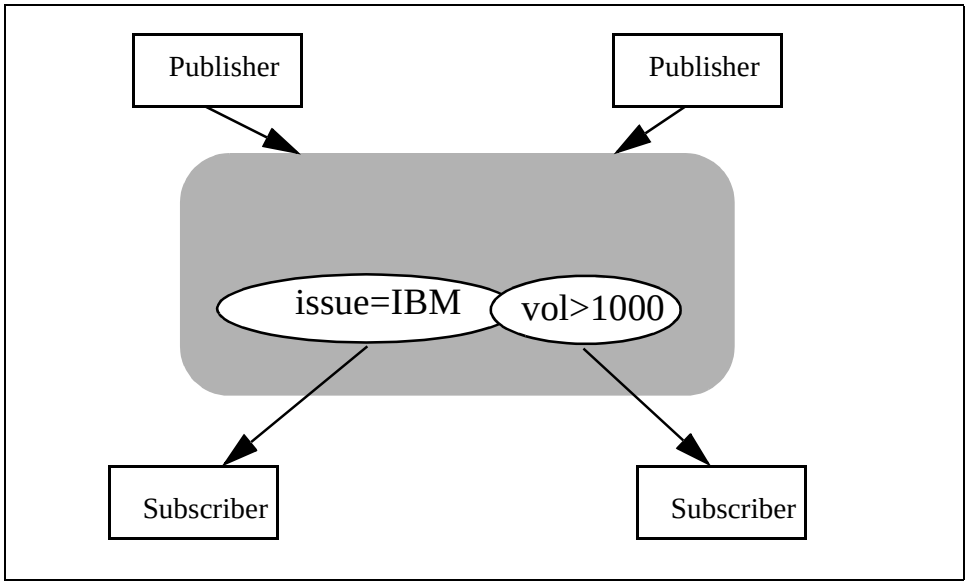


**FIGURE 2.** Content-based publish/subscribe systems scale with application growth.




## Industry Standard API

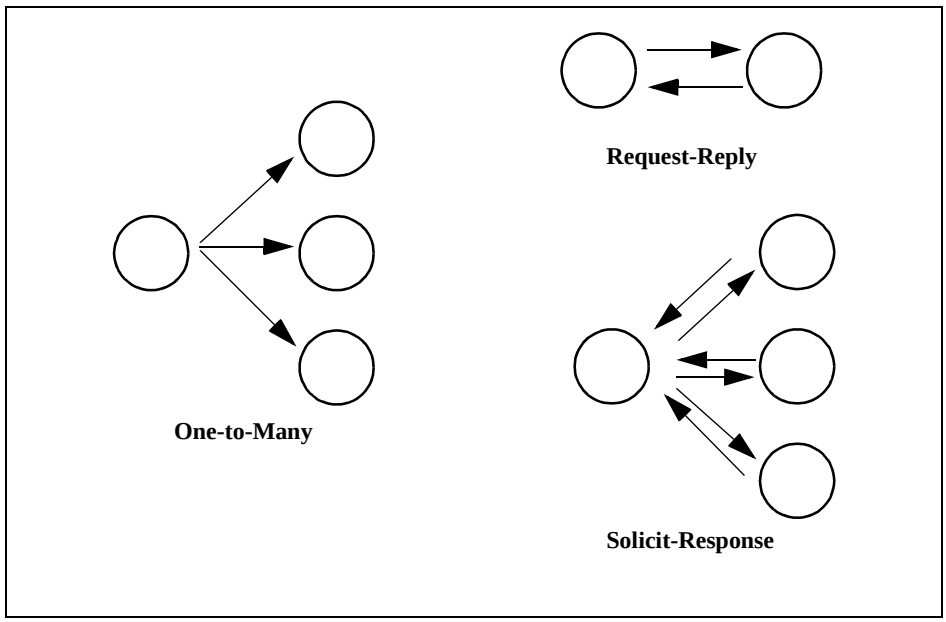
Gryphon implements the publish/subscribe portion of the industry-standard Java Message Services (JMS) API. JMS is part of Java 2, Enterprise Edition. It specifies how clients establish connections to a publish/subscribe service, subscribe to messages, publish messages, and build messages. Because JMS is part of Java 2, and the JMS package itself is easily deployed with Java 1.1, Gryphon clients can be deployed on any platform supported by a JVM, including web browsers.

JMS provides a large degree of flexibility in client subscriptions. Clients may subscribe using both a topic identifying messages tagged with the topic by a publisher, and using filters that select messages based on their content. The content filter language is based on the WHERE syntax of SQL 92.

Gryphon uses a patented matching engine to provide high-speed content filtering comparable to topic-only approaches. Gryphon also organizes topics into a hierarchy and supports wildcards to allow subscribers maximum flexibility in their topic subscriptions. The combination of hierarchical topics and high-speed content filtering provides the flexibility necessary to allow applications to evolve beyond their initial design.

JMS also supports a range of messaging interaction patterns as shown in Figure 3. Traditionally, publish/subscribe messaging supports one-to-many delivery of data. In addition, JMS enables other interaction patterns by allowing cli-

**FIGURE 3.** Messaging Interactions Possible within Gryphon




ents to create unique topics. Using unique topics, JMS applications may use request-reply messaging. Furthermore, request-reply is easily extended with the one-to-many delivery of publish/subscribe to provide solicit-response. Solicit-response allows clients to use publish/subscribe features to make an *advertisement* to which many other clients may respond privately.

## Scalability & Reliability

The key to Gryphon's scalability and reliability is the configuration of a group of Gryphon message brokers into a *multibroker* network. A Gryphon multibroker network may accommodate growth in client load by simply adding additional broker machines. Gryphon provides a rich set of configuration options allowing multibroker deployments to exploit underlying network configurations and thereby ensures linear scalability.

The basic building block of a multibroker configuration is the *cell:* a group of fully connected servers that provide scalability at the cluster level. Cells may then be linked together for geographic scaling through *link bundles* that provide redundant connections between cells. The redundancy of link bundles provides load-balancing and fault-tolerance not provided by gateway based approaches. Gryphon's internal protocols ensure cycles are avoided and messages are routed around failures.

## Security

Gryphon provides a full set of security features critical for applications deployed over public networks. The basis of Gryphon security is a *menu* of client authentication options: simple (telnet-like) password, mutual password authentication, asymmetric password-certificate SSL authentication, and symmetric certificate SSL. Clients negotiate an authentication protocol with a server upon connection. Access Control Lists (ACLs) may limit to which topic an authenticated client may publish messages or subscribe. Both positive and negative access controls may be specified for any region of the topic hierarchy.

Gryphon also supports encryption and cryptographic integrity of messages. Gryphon computes a "quality of protection" for each message based on its topic to ensure that encryption or cryptographic hashing is applied to only those messages where it is necessary. Furthermore, Gryphon uses optimized end-to-end encryption techniques to allow content-based routing without excessive cost for cryptographic operations.

To support a wide range of deployments, Gryphon provides tunnelling support for SOCKS and HTTP. Tunnelling is provide both from clients to servers and between servers. Tunnelling from clients allows public Gryphon servers to be reached from within intranets. In addition, by using tunnelling between servers, clients may be directed to a Gryphon cluster within a firewall, reducing the load on the firewall. The firewall is burdened only with connections from the server within the firewall to the public Gryphon network. Given this capability, Gryphon servers may be placed anywhere a HTTP proxy may be placed with the expected performance enhancement.

## Gryphon History

In 1997, the Gryphon project was initiated at the IBM T. J. Watson Research Center to support the next generation of web applications. These applications were predicted to go beyond the current generation of pull-based applications to provide data in real-time to thousands of clients world-wide. A scalable, reliable, and secure content-based publish/subscribe system that could be deployed over *public* networks would be needed to support these applications. In 1999, Gryphon was deployed for the first of these new applications through a joint effort between IBM Research and IBM Global Services. Sports scores were delivered in real-time off the web sites for the US Tennis Open, the Ryder Cup, and the Australian Open. Gryphon was also used as a core part of the Sydney Olympics Intranet, providing real-time monitoring and statistics data.

# LEO: An autonomic query optimizer for DB2

by V. Markl
G. M. Lohman
V. Raman

Structured Query Language (SQL) has emerged as an industry standard for querying relational database management systems, largely because a user need only specify what data are wanted, not the details of how to access those data. A query optimizer uses a mathematical model of query execution to determine automatically the best way to access and process any given SQL query. This model is heavily dependent upon the optimizer's estimates for the number of rows that will result at each step of the query execution plan (QEP), especially for complex queries involving many predicates and/or operations. These estimates rely upon statistics on the database and modeling assumptions that may or may not be true for a given database. In this paper, we discuss an autonomic query optimizer that automatically self-validates its model without requiring any user interaction to repair incorrect statistics or cardinality estimates. By monitoring queries as they execute, the autonomic optimizer compares the optimizer's estimates with actual cardinalities at each step in a QEP, and computes adjustments to its estimates that may be used during future optimizations of similar queries. Moreover, the detection of estimation errors can also trigger reoptimization of a query in mid-execution. The autonomic refinement of the optimizer's model can result in a reduction of query execution time by orders of magnitude at negligible additional run-time cost. We discuss various research issues and practical considerations that were addressed during our implementation of a first prototype of LEO, a LEarning Optimizer for DB2® (Database 2™) that learns table access cardinalities and for future queries corrects the estimation error for simple predicates by adjusting the database statistics of DB2.

The remarkable growth of the relational database management systems (DBMS) industry over the last two decades can be largely attributed to the standardization of its Structured Query Language, SQL. SQL is a declarative language, that is, it requires the user to specify only *what* data are wanted, leaving to the *query optimizer* of the DBMS the difficult problem of deciding *how* best to access and process the data. For a given SQL query, there may be many different ways to access each table that is referenced, to join those tables, and, because the join operation is commutative, to order those joins and perform other operations necessary to complete the query. Hence, there may be hundreds or even thousands of possible ways to process a given query correctly. For example, suppose the SQL query is:

```
SELECT name, age, salary
FROM employees
```




0018-8670/03/$5.00 © 2003 IBM

```
WHERE age > 60 AND city = 'SAN JOSE' AND
    salary < 60,000
```

This query asks for the name, age, and salary of each employee who is over age 60, lives in San Jose, and makes less than $60,000 annually in salary. Each filtering condition in the WHERE clause that is joined by AND is called a *predicate*. Since this query references only one table, there are no choices of join order or join method, yet the optimizer still may consider many possible ways for the DBMS to process this simple query. It can always sequentially scan all the rows in the table and apply each predicate to each row. Or, if the appropriate indexes exist, it could exploit one or more indexes to access only the rows satisfying one or more of the predicates. For example, an index on age would permit accessing only those rows where the value of age is greater than 60 and then applying the other predicates (on city and salary). Alternatively, an index on city would limit the access to those rows having city equal to "San Jose" and subsequently applying the other predicates (on age and salary) to those rows retrieved. Alternatively, indexes on multiple columns, for example a combined index on city and age, or a combined index on city and salary, might be exploited, if they existed, or strategies combining any of the indexes discussed above (so-called "index ANDing"). Which strategy might be preferable depends upon the characteristics of the database, the availability of various indexes, and how selective each predicate is, that is, how many rows are satisfied by each condition.

Most modern query optimizers determine the best query execution plan (QEP, or simply *plan*) for executing an SQL query by mathematically modeling the execution cost for each of many alternative QEPs and choosing the one with the lowest estimated cost. The execution cost is largely dependent upon the number of rows that will be processed by each operator in the QEP, so the optimizer first estimates this incrementally as each predicate is applied. Estimating the *cardinality* (i.e., number of rows) of a result, after one or more of the predicates have been applied, has been the subject of much research for over 20 years.[1–5] To avoid accessing the database when optimizing queries, this estimate typically begins with statistics of database characteristics, specifically, the number of rows for each table. The cardinality of each intermediate result is derived incrementally by multiplying the base table's cardinality by a *filter factor*—or *selectivity*—for each predicate in the query, which is derived from statistics on the columns affected by that predicate, such as its number of distinct values or a histogram of its distribution. The selectivity of a predicate P effectively represents the probability that any row in the table will satisfy P, and that selectivity depends upon the characteristics of the database. For example, in the query above, the predicate on city might be quite selective if the database were a worldwide database of a large, multinational company, but it might be a lot less selective if the database contained all the employees of some small start-up firm centered in San Jose. For the latter case, the predicates on age and/or salary would be more selective. The optimizer would tend to favor QEPs that could exploit indexes to apply the most selective predicates and QEPs that utilize simple table scans if there were no indexes, or if the optimizer estimated that most employees would satisfy all of the predicates. In DB2* (Database 2*), the choice of QEP is based solely upon the detailed cost estimate for each of the competing plans, and not upon such simplistic heuristics.

When there are multiple tables in the FROM clause of the query, the number of alternative strategies considered by the optimizer increases exponentially, because the myriad choices mentioned above are compounded with additional decisions about the order in which tables are joined and the method by which they are joined. DB2, for example, supports three major types of join method, and there are several variants within each of these. For a two-table join with a handful of predicates, the DB2 optimizer might consider over a hundred different plans; for six tables, the number of plans could be well over a thousand! The DB2 optimizer considers all of these alternatives automatically for the user, who is not even aware that it is going on!

Although query optimizers do a remarkably good job of estimating both the cost and the cardinality of most queries, many assumptions underlie this mathematical model. Examples of these assumptions include: currency of information, uniformity, independence of predicates, and a principle of inclusion.

*Currency of information:* Updating the statistics each time a row is updated or deleted would create a locking bottleneck in the system catalogs, where statistics are stored. It is difficult or impossible to calculate some statistics incrementally, such as the number of distinct values for each column, and so it is common for statistics to be recomputed periodically as a user-invoked batch operation (called *RUNSTATS* in DB2). Despite this, the optimizer assumes that the statistics reflect the current state of the database, that

is, that the database characteristics are relatively stable, and it relies upon the user to know when any table has changed enough to warrant the expensive recollection of statistics.

*Uniformity:* Although many products use histograms to deal with skew in the distribution of values for "*local*" selection predicates (on columns within a single table), we are unaware of any available product that exploits them for *join* predicates, that is, those relating columns in multiple tables. Thus for join predicates, the query optimizer still relies on the assumption of uniformity.

*Independence of predicates:* Selectivities for each predicate are calculated individually and multiplied together, essentially assuming the predicates are statistically independent of each other, even though the underlying columns may be related, for example by a functional dependency. While multidimensional histograms address this problem for local predicates, they have never been applied to join predicates, aggregation, and so on. Applications common today have hundreds of columns in each table and thousands of tables, making it impossible to know on which subset(s) of columns to maintain multivariate statistics.

*Principle of inclusion:* The selectivity for a join predicate X.a = Y.b is typically defined to be $1/\max\{|a|, |b|\}$, where |b| denotes the number of distinct values of column b. This implicitly assumes the "principle of inclusion," that is, that each value of the smaller domain has a match in the larger domain. Fortunately, this assumption is frequently true for the most common joins between a *primary key* to a table (e.g., a product number in the Products table) and a reference to that key (a *foreign key*) in another table (e.g., the Orders table).

When these assumptions are invalid, significant errors in the cardinality—and hence cost—estimates result, causing suboptimal plans to be chosen. From the authors' experience, the primary cause of major modeling errors is the cardinality estimate on which costs depend. Cost estimates might be off by 10 or 15 percent, at most, for a given cardinality, but cardinality estimates can be off by orders of magnitude when their underlying assumptions are invalid or uncertain. Although there has been considerable success in using histograms to detect and correct for data skew,[6–8] and in using sampling to gather up-to-date statistics,[9,10] there has been to date no comprehensive approach to correcting all modeling errors, regardless of origin.

In this paper we describe our approach toward autonomic query optimization for overcoming modeling errors and incorrect statistics, which has led to the prototype of a LEarning Optimizer (LEO) for DB2.[11] LEO learns from any modeling mistake, at any point in a QEP, by automatically validating its estimates against actual cardinalities for a query. Determining at what point in the plan the significant errors occurred then allows for reoptimizing the query at this point[12] and adjusting its model dynamically to better optimize future queries. Over time, this feedback method amasses experiential information that augments and adjusts the database statistics for the part of the database that enjoys the most user activity. Not only does this information enhance the quality of the optimizer's estimates, but it can also suggest where statistics gathering should be concentrated and can even supplant the need for statistics collection.

## A learning optimizer

This section describes the architecture of LEO, an *autonomic optimizer* that observes actual query execution and uses actual cardinalities to autonomically validate and refine the estimates from its model and to reoptimize the current query, without requiring user intervention. In the following sections we discuss the two essential functions of LEO: deferred learning for future queries and progressive optimization of the query currently under execution.

**Deferred feedback-based learning.** Deferred learning exploits empirical results from actual executions of queries to validate the optimizer's model incrementally, deduce what part of the optimizer's model is in error, and compute adjustments to the optimizer's model for future query optimizations. Deferred learning with LEO works under the assumption that future queries will be similar to previous queries, that is, they will share one or several predicates. Our LEO prototype currently corrects the statistics for tables (which may be out of date) and estimates the selectivity of individual predicates in this way.

The LEO feedback loop is comprised of four steps, as seen in Figure 1: monitoring, analysis, feedback, and feedback exploitation. At query compilation time, the monitoring component saves the cardinality estimates derived by the optimizer for the best (i.e., least-cost) plan, and during query execution

Figure 1 Deferred learning

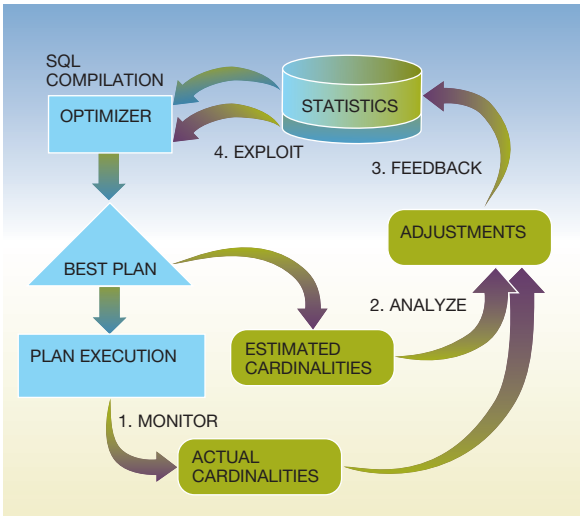
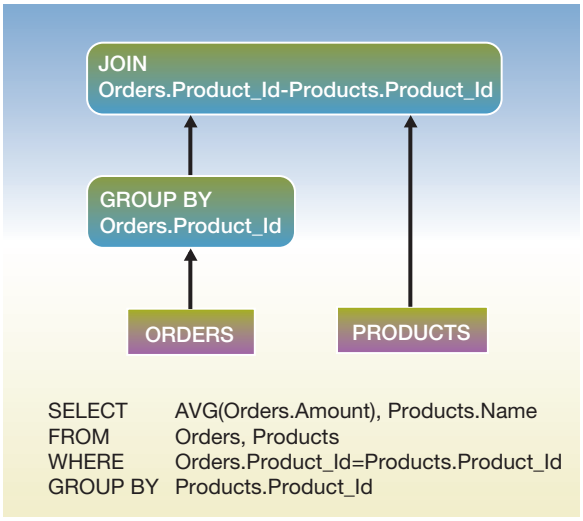



Figure 2 A query plan that can be reoptimized dynamically




saves the actual cardinalities observed for that plan. The analysis component uses the information thus learned to identify modeling errors and compute corrective adjustments. This analysis is a stand-alone process that may be run separately from the database server and even on another system. The feedback component modifies the catalog statistics of the database according to the learned information. The exploitation component closes the feedback loop by using the learned information in the system catalog to provide adjustments to the query optimizer's cardinality estimates.

The four components can operate independently, but form a consecutive sequence that constitutes a continuous learning mechanism by incrementally capturing plans, monitoring their execution, analyzing the monitor output, and computing adjustments to be used for future query compilations. This mechanism enables deferred learning, since only future queries will benefit from the feedback.

The deferred learning mechanism has been implemented in a prototype using DB2 Universal Database* (UDB) for Linux**, UNIX**, and Windows**. Experiments with the protype[18] showed the monitoring overhead to be below 4 percent of the total query execution time, whereas performance may improve by orders of magnitude, particularly when the optimizer learns that a bulk join method should be used instead of a nested-loop join, due to a large input cardinality.

**Immediate feedback-based learning.** The monitored cardinalities need not be used for subsequent queries alone. If the actual cardinalities are significantly different from the estimated cardinalities, the chosen query plan could be highly suboptimal. As part of the LEO project, we are currently investigating how to use this knowledge immediately by dynamically reoptimizing the current query and changing its execution plan, if all of the rows for an intermediate result are materialized before proceeding at any point in the plan.

Generally, response time and memory are optimized if each row is processed completely and returned to the user in a *pipelined plan*. But occasionally, the rows of an intermediate result must be fully materialized, either as a sorted or unsorted temporary table (TEMP), which we call a *materialization point*. TEMPs afford a natural opportunity to count the number of rows and possibly to reoptimize the plan before any rows are returned to the user, thereby avoiding returning duplicate rows that are caused by restarting the query. However, two important issues arise:

- Since reoptimization involves a cost, when is it worthwhile?
- How can we reoptimize efficiently?

We address the first question in the subsection, "When to reoptimize." For the second question, the

easiest solution is to simply rerun the query "from scratch" under a new plan. However, this would waste all the (possibly substantial) work done up to the materialization point, which was saved in the TEMP. In most cases, it is preferable for the reoptimized plan to avoid having to redo that work by instead accessing that TEMP in the reoptimized plan.

For example, Figure 2 shows a query plan for a simple two-table join that groups/aggregates the Orders table by Product_Id before the join. The sort that may be needed to accomplish this aggregation must materialize its entire input before proceeding and thus constitutes a TEMP. Since most aggregations can be performed incrementally as the rows are sorted, the TEMP will, at its conclusion, contain the GROUP BY result. The optimal join algorithm (nested loop join, hash join, or merged join) for subsequently joining Orders and Products depends crucially on the size of this GROUP BY result. The query optimizer could choose a suboptimal join algorithm if it under- or over-estimates the size of this result.

However, during query execution, the optimizer can monitor the size of the GROUP BY result, and reoptimize in case of severe estimation errors, for example, by changing the join algorithm if needed. Such reoptimization becomes more complex for more elaborate query plans with multiple materialization points. Reference 12 suggests encapsulating TEMPs as tables and converting the remaining portion of the query plan after the TEMPs into an SQL query, which can then be resubmitted to the query optimizer. Unfortunately, this approach has two problems. First, it may not be optimal to reuse a TEMP as is. In cases where the size of the TEMP is much larger than expected, the optimal plan might be to reuse only a part of the TEMP, or even ignore the TEMP completely, in favor of a totally new plan that directly uses the base tables. Second, the remaining portion of the plan beyond the TEMP may not always be expressible as an SQL statement, especially if it contains update operations, which are fed from subqueries.

A better alternative is not to encapsulate the TEMPs, but instead to define them as *materialized views*[13] (known in DB2 as *Automatic Summary Tables* or ASTs[14]) and expose their definition to the query optimizer. The optimizer can then rely on standard view-matching techniques[14,15] to identify TEMPs that are worthwhile to reuse. The cost of reoptimization using additional materialized views is almost identical to the cost of optimizing the original query, since the optimizer only has to investigate one alternative intermediate table access method per materialized view.

Moreover, once TEMPs are defined as materialized views, there is no reason to limit their use to the current query only. All subsequent queries can potentially exploit materialized TEMPs, just as they exploit user-defined materialized views. Of course, this approach could lead to an avalanche of such views, so that the query engine would have to periodically delete rarely used ones; this is akin to the materialized view selection problem.[16]

## Research issues in autonomic query optimization

Our initial prototype of LEO has uncovered a number of challenging research problems that require solutions for any practical application of the optimizer in a product. We now discuss these problems and possible approaches to their solution.

**Stability and convergence.** A cardinality model refined by feedback has to take incomplete information into account. While some cardinalities may be deduced from query feedback—constituting *hard facts*—others are derived from statistics and modeling assumptions—forming *uncertain knowledge*. The learning rate of the system is largely dependent on the workload and the accuracy of statistics and assumptions.

Assuming independence of predicates, when in fact the data are correlated, usually results in underestimation of the cardinalities of the intermediate results, which are used by the optimizer when determining the cost of a QEP. This underestimation will cause the optimizer to prefer a plan based on uncertain knowledge over one based on hard facts. The underestimation of cardinalities can result in a complete exploration of the search space; the system will converge only after trying out and learning about all QEPs that contain underestimation. Overestimation, however, may result in a local minimum (i.e., a suboptimal QEP); the optimizer will prefer other QEPs over a QEP with overestimates. Hence overestimates are unlikely to ever be discovered or corrected.

To reach a reasonable form of stability, the autonomic optimizer should initially use an exploratory mode, for example, before going into production. This mode will initially involve more risks by choosing promising QEPs based on uncertain knowledge,

thus validating the model and gathering hard facts about data distribution and workload. A second operational mode will be biased toward QEPs that are based on experience. This mode favors QEPs based on hard facts over slightly cheaper QEPs based on uncertain knowledge. The transition between the modes would be gradual, resembling simulated annealing[17] methods in machine learning.

To overcome the local optimum caused by overestimation, it is necessary to explore uncertain knowledge used for presumably suboptimal, but promising QEPs, for example, by synchronous or asynchronous sampling.[10]

**Detecting and exploiting correlation.** In practical applications, data are often highly correlated. In a car database, for instance, the selectivity of the conjunction (*make* = "Honda" and *model* = "Accord") is *not* correctly derived by multiplying the individual selectivities of *make* = "Honda" and *model* = "Accord," because the columns *make* and *model* are correlated—only Honda makes an Accord model. Since correlation constitutes a violation of the independence assumption, selectivity estimates for predicates involving correlation can be off by orders of magnitude in state-of-the-art query optimizers. With our approach, we have the opportunity to detect and correct such errors.

Correlations pose many challenges. First, there are many types of correlation, ranging from functional dependencies between columns, especially referential integrity, to more subtle and complex cases, such as an application-specific constraint that an item is supplied by at most 20 suppliers. Second, correlations may involve more than two columns, and hence more than two predicates in a query, with subsets of those columns having varying degrees of correlation. Third, a single query can only provide evidence that two or more columns are correlated for specific values. For complex queries involving several predicates, isolating which subsets of predicates are correlated and the degree of correlation can be extremely difficult. Another difficult research problem is to generalize correlations from specific values to relationships between columns: How many different values from executing multiple queries having predicates on the same columns are required to safely conclude that those columns are, in general, correlated, and to what degree? Instead of waiting for that many queries to execute, correlation detection could instead identify promising combinations of columns—even from different tables—on which the statistics utility would then collect multidimensional histograms. In addition, the observed information can be used to pinpoint errors in the cardinality model, populate the database statistics, or to adjust the erroneous estimates by creating an additional layer of statistics.

**When to reoptimize.** As discussed in the subsection, "Immediate feedback-based learning," immediate learning can change the plan for a query at run time, when the actual cardinalities are significantly different from the estimated cardinalities. But the new plan could itself be quite expensive, if it cannot make use of prior TEMPs efficiently. The optimizer will find this out during reoptimization, but the cost of reoptimization could itself be significant. Therefore it is crucial to determine, *without reoptimizing*, when it will be worthwhile to reoptimize.

Reference 12 uses the difference between the estimated and actual cardinalities as a heuristic to determine whether to reoptimize. However the question is not how inaccurate the optimizer's estimate is; it is whether the plan is suboptimal under the new cardinalities and whether the cost difference is enough to pay for the reoptimization. One heuristic looks at the nature of the plan operators and decides whether a change in the input cardinality for an operator is likely to make the operator suboptimal. Alternatively, the optimizer can be enhanced to pick not only the optimal plan, but also the range of selectivities for each predicate within which the plan is optimal. This prediction of the sensitivity of any plan to any one parameter is extremely hard, because of nonlinearities in the cost model.

We also need to limit the number of reoptimization attempts for a single query, because the convergence problem of the subsection, "Stability and convergence" is even more serious here. We do not want the query execution to get into a long loop where it repeatedly tries out all alternative plans before making progress.

**Learning other information.** Learning and adapting to a dynamic environment is not restricted to cardinalities and selectivities. Using a feedback loop, many costs and parameters currently estimated by the optimizer can be made self-validating. For example, the dominant aspect of cost, the number of physical I/Os, is currently estimated probabilistically from estimated hit ratios, assuming each application gets an equal share of the buffer pool. The optimizer could validate these estimates by observing actual

I/Os, actual hit ratios, and/or actual times to access tables for a given plan. Another example is the maximum amount of memory allocated to perform a particular sort in a plan. If the DBMS detected by query feedback that a sort operation could not be performed in main memory, it could adjust the sort heap size to avoid external sorting for future sort operations.

Feedback is not limited to services and resources consumed by the DBMS, but also extends to the applications that the DBMS serves. For example, the DBMS could measure how many of the rows in a query's result are actually consumed by each application and optimize each query's performance for just that portion of the result, for example, by effectively appending the OPTIMIZE FOR <n> ROWS clause of SQL to that query. Similarly, feedback from executions could be used to automatically set many configuration parameters for shared resources that are currently set manually. Physical parameters such as the network rate, disk access time, and disk transfer rate are used to weight the contribution of these resources to plan costs, and are usually considered to be constant after an initial set-up. However, setting these parameters using measured values is more autonomic and more accurately captures the effective rate. In the same way, the allocation of memory among different buffer pools, the total sort heap, and so on, can be tuned automatically according to hit ratios that were recently observed.

## Practical considerations

In the process of implementing LEO, several practical considerations also needed to be addressed.

**The Hippocratic oath: "Do no harm!"** The overall goal of an autonomic optimizer is to improve query performance by adjusting an existing model based upon previously executed queries. Ideally, this adjusted model provides a better decision basis for selecting the best execution plan for a query. However, this learned knowledge must be arrived at extremely conservatively: we should not make hasty conclusions based upon inconclusive or spotty data. In critical applications, stability and reliability of query processing are often favored over optimality with occasional unpredictable behavior. If adjustments are immediately taken into account for query optimization, even on a highly dynamic database, the same query may generate a different execution plan each time it is issued, and thus may result in thrashing of execution plans. This instability can be avoided if reoptimization of queries takes place after the learned knowledge has converged to a fixed point or has reached a defined threshold of reliability.

**Consistency between statistics.** DB2 collects statistics for base tables, columns, indexes, functions, and tablespaces, many of which are mutually interdependent. DB2 permits users to update the statistics in the catalogs and performs checks for inconsistencies in such updates. An autonomic optimizer must similarly ensure the consistency of these interdependent statistics when adjusting any of them. For example, the number of rows of a table determines the number of disk pages used for storing those rows. When adjusting the number of rows of a table, we must consequently ensure consistency with the number of pages of that table—for example, by adjusting this figure as well—or else plan choices may be biased, depending on which plan uses which statistic. Similarly, the consistency between index and table statistics has to be preserved, since there may be interdependencies between the number of distinct values of a column and the number of rows in a table. However, an increase in the number of rows will not always result in an increase in the number of distinct values: although subsequent inserts are likely to alter the number of distinct values for a *date* column, this is very unlikely for a column like *sex* that can only assume the values *male* or *female*, regardless of the number of rows.

**Adjustments vs database statistics.** An autonomic optimizer is not a replacement for database statistics, but rather a complement to them. Statistics are collected uniformly across the database, to prepare for any possible query. Feedback gives the greatest improvement to the modeling of queries that are either repetitive or are similar to earlier queries, that is, queries for which the optimizer's model exploits the same statistical information. Feedback extends the capabilities of the RUNSTATS utility by gathering information on derived tables (e.g., the result of several joins) and gathering more detailed information than RUNSTATS might. Over time, the optimizer's estimates will improve most in regions of the database that are queried most. However, for correctly estimating the cost of previously unanticipated queries, the statistics collected by RUNSTATS are necessary, even in the presence of query feedback.

## Conclusions

Although today's query optimizers autonomically determine the best way to process a declarative SQL

query (one which specifies only what data are wanted), they do so using a complex mathematical model having many inherent assumptions and parameters. The ideas on autonomic query optimization outlined in this paper have led to the implementation of LEO, DB2's LEarning Optimizer. By self-validating these assumptions and parameters using feedback garnered from earlier executions, LEO provides a major step forward in improving the quality of query optimization and reducing the need for "tuning" of problem queries, a major contributor to cost of ownership. Our current LEO prototype enables deferred learning of table access cardinalities and simple predicates,[18] demonstrating significant performance improvements and a low monitoring overhead of below 4 percent of the total query execution time.

Our future work includes completing the LEO prototype for deferred learning, aggregating and summarizing the observed information, finding conclusive ways to discern and generalize occurrences of correlation among predicates, measuring the benefit of using LEO on a realistic set of user queries, and extending LEO's approach to parameters other than cardinality. In addition, we are carrying out a prototype implementation of immediate learning in order to analyze and validate the performance of this progressive query optimization approach.

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of Linus Torvalds, The Open Group, or Microsoft Corporation.

## Cited references


1. P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price, "Access Path Selection in a Relational Database Management System," *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Boston, MA, May 1979, ACM, New York (1979), pp. 23–24.
2. A. Van Gelder, "Multiple Join Size Estimation by Virtual Domains," *Proceedings of the Twelfth ACM Symposium on Principles of Database Systems* (May 1993), pp. 180–189.
3. A. N. Swami and K. B. Schiefer, "On the Estimation of Join Result Sizes," 4th International Conference on Extending Database Technology (March 1994), pp. 287–300.
4. R. Ahad, K. V. B. Rao, and D. McLeod, "On Estimating the Cardinality of the Projection of a Database Relation," *ACM Transactions on Database Systems* **14**, No. 1, pp. 28–40 (1989).
5. C. Lynch, "Selectivity Estimation and Query Optimization in Large Databases with Highly Skewed Distributions of Column Values," *Proceedings of the 14th International Conference on Very Large Databases* (August 1988), pp. 240–251.
6. Y. E. Ioannidis and S. Christodoulakis, "On the Propagation of Errors in the Size of Join Results," *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Denver, CO, May 1991, ACM, New York (1991), pp. 268–277.
7. V. Poosala, Y. Ioannidis, P. Haas, and E. Shekita, "Improved Histograms for Selectivity Estimation of Range Predicates," *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Montreal, Canada, June 1996, ACM, New York (1996), pp. 294–305.
8. V. Poosala and Y. Ioannidis, "Selectivity Estimation Without the Attribute Value Independence Assumption," *Proceedings of the 23rd International Conference on Very Large Databases* (VLDB 1997).
9. P. Haas, J. Naughton, S. Seshadri, and A. Swami, *Selectivity and Cost Estimation for Joins Based on Random Sampling*, Research Report RJ-9577, IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598 (1993).
10. T. Urhan, M. J. Franklin, and L. Amsaleg, "Cost-Based Query Scrambling for Initial Delays," *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Seattle, WA, June 1998, ACM, New York (1998), pp. 130–141.
11. M. Stillger, G. Lohman, V. Markl, and M. Kandil, "LEO—DB2's Learning Optimizer," *Proceedings of the 27th International Conference on Very Large Databases* (September 2001), pp. 19–28.
12. N. Kabra and D. DeWitt, "Efficient Mid-Query Re-Optimization of Sub-Optimal Query Execution Plans," *Proceedings of the ACM SIGMOD International Conference on Management of Data* (June 1998), pp. 106–117.
13. N. Roussopoulos, "Materialized Views and Data Warehouses," *SIGMOD Record* **27**, No. 1, 21–26, ACM, New York (1998).
14. M. Zaharioudakis, R. Cochrane, G. Lapis, H. Pirahesh, and M. Urata, "Answering Complex SQL Queries Using Automatic Summary Tables," *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Dallas, TX, May 2000, ACM, New York (2000), pp. 105–116.
15. S. Chaudhuri, R. Krishnamurthy, S. Potamianos, and K. Shim, "Optimizing Queries with Materialized Views," *Proceedings of the Eleventh International Conference on Data Engineering* (March 1995), pp. 190–220.
16. R. Chirkova, A. Y. Halevy, and D. Suciu, "A Formal Perspective on the View Selection Problem," *Proceedings of the 27th International Conference on Very Large Databases* (September 2001), pp. 59–68.
17. S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by Simulated Annealing," *Science* **220**, No. 4598, 671–680 (May 1983).
18. V. Markl and G. M. Lohman, "Learning Table Access Cardinalities with LEO," *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Madison, WI, June 2002, ACM, New York (2002), p. 613.


*Accepted for publication October 11, 2002.*

**Volker Markl** *IBM Research Division, Almaden Research Center, 650 Harry Road, San Jose, California 95120 (electronic mail: marklv@us.ibm.com).* Dr. Markl is a research staff member in the Advanced Database Solutions Department at the Almaden Research Center in San Jose, California, conducting research in query optimization, indexing, and self-managing databases. Dr. Markl is spearheading the LEO project at IBM, an effort in autonomic computing with the goal of creating a self-tuning optimizer for DB2. Dr. Markl holds a Ph.D. degree and M.S. degree in computer science from Technische Universität München, as well as a degree in business administration from Universität Hagen, Germany. In his earlier professional career, Dr. Markl coinvented and developed the enabling indexing technology for the relational database management system TransBase HyperCube,

which was awarded the European Information Society Technology Prize in 2001 by the European Commission.

**Guy M. Lohman** *IBM Research Division, Almaden Research Center, 650 Harry Road, San Jose, California 95120 (electronic mail: lohman@almaden.ibm.com).* Dr. Lohman is manager of advanced optimization in the Advanced Database Solutions Department at the Almaden Research Center in San Jose, California, and has 20 years of experience in relational query optimization. He is the architect of the Optimizer of the DB2 Universal Database (UDB) for Linux, UNIX, and Windows, and was responsible for its development in Versions 2 and 5. During that period, Dr. Lohman also managed the overall effort to incorporate into the DB2 UDB product the Starburst compiler technology that was prototyped at the Almaden Research Center. More recently, he was a coinventor and designer of the DB2 Index Advisor, and cofounder of the DB2 SMART (Self-Managing And Resource Tuning) project, part of IBM's autonomic computing initiative. In 2002, Dr. Lohman was elected to the IBM Academy of Technology. His current research interests involve query optimization and self-managing database systems.

**Vijayshankar Raman** *IBM Research Division, Almaden Research Center, 650 Harry Road, San Jose, California 95120 (electronic mail: ravijay@us.ibm.com).* Dr. Raman is a research staff member at the Almaden Research Center, with a focus on data management issues in grid computing, and on adaptive query optimization. He is also interested in algorithmic mechanism design, and in data cleaning and integration. Dr. Raman graduated from the University of California, Berkeley, in 2001 with a Ph.D. degree in computer science, specializing in database management systems. His research resulted in a dozen refereed papers in international conferences and journals, one of which was selected as one of the best papers at the 25th International Conference on Very Large Databases. One component of his research has evolved into the Potter's Wheel open source data cleaning software. Dr. Raman was awarded a Microsoft Fellowship during his graduate study. He also won an AT&T Asia-Pacific Leadership Award for achievements during his undergraduate study at the Indian Institute of Technology, Madras.

# LEO – DB2's LEarning Optimizer

**Michael Stillger, Guy Lohman, Volker Markl**

**IBM Almaden Research Center, San Jose, CA, 95139 USA**

msstillger@aol.com, {lohman, marklv}@almaden.ibm.com

**Mokhtar Kandil**

**IBM Canada Ltd. Development Laboratory, Toronto, ON, Canada**

mkandil@ca.ibm.com

**Abstract**

Most modern DBMS optimizers rely upon a cost model to choose the best query execution plan (*QEP*) for any given query. Cost estimates are heavily dependent upon the optimizer's estimates for the number of rows that will result at each step of the QEP for complex queries involving many predicates and/or operations. These estimates, in turn, rely upon statistics on the database and modeling assumptions that may or may not be true for a given database. In this paper we introduce LEO, DB2's LEarning Optimizer, as a comprehensive way to repair incorrect statistics and cardinality estimates of a query execution plan. By monitoring previously executed queries, LEO compares the optimizer's estimates with actuals at each step in a QEP, and computes adjustments to cost estimates and statistics that may be used during future query optimizations. This analysis can be done either on-line or off-line on a separate system, and either incrementally or in batches. In this way, LEO introduces a feedback loop to query optimization that enhances the available information on the database where the most queries have occurred, allowing the optimizer to actually learn from its past mistakes. Our technique is general and can be applied to any operation in a QEP (not just selection predicates on base tables), including joins, derived results after several predicates have been applied, and even to DISTINCT and GROUP-BY operators. As shown by performance measurements on a 10 GB TPC-H data set, the runtime overhead of LEO's monitoring is insignificant, whereas the potential benefit to response time from more accurate cardinality and cost estimates can be orders of magnitude.

# 1 Introduction

Most modern query optimizers for relational database management systems (*DBMSs*) determine the best query execution plan (*QEP*) for executing an SQL query by mathematically modeling the execution cost for each plan and choosing the cheapest QEP. This execution cost is largely dependent upon the number of rows that will be processed by each operator in the QEP. Estimating the number of rows – or *cardinality* – after one or more predicates have been applied has been the subject of much research for over 20 years [SAC+79, Gel93, SS94, ARM89, Lyn88]. Typically this estimate relies on statistics of database characteristics, beginning with the number of rows for each table, multiplied by a *filter factor* – or *selectivity* – for each predicate, derived from the number of distinct values and other statistics on columns. The selectivity of a predicate P effectively represents the probability that any row in the database will satisfy P.

While query optimizers do a remarkably good job of estimating both the cost and the cardinality of most queries, many assumptions underlie this mathematical model. Examples of these assumptions include:

- *Currency of information:* The statistics are assumed to reflect the current state of the database, i.e. that the database characteristics are relatively stable.
- *Uniformity:* Although histograms deal with skew in values for "local" selection predicates (to a single table), we are unaware of any available product that exploits them for joins.
- *Independence of predicates:* Selectivities for each predicate are calculated individually and multiplied together, even though the underlying columns may be related, e.g. by a functional dependency. While multi-dimensional histograms address this problem for local predicates, again they have never been applied to join predicates, aggregation, etc. Applications common today have hundreds of columns in each table and thousands of tables, making it impossible to know on which subset(s) of columns to maintain multi-dimensional histograms.
- *Principle of inclusion:* The selectivity for a join predicate X.a = Y.b is typically defined to be $1/\max\{|a|, |b|\}$, where |b| denotes the number of distinct values of column b. This implicitly assumes the "principle of inclusion", i.e. that each value of the smaller domain has a match in the larger domain (which is frequently true for joins between foreign keys and primary keys).

When these assumptions are invalid, significant errors in the cardinality – and hence cost -- estimates result, causing sub-optimal plans to be chosen. From the authors' experience, the primary cause of major modeling errors is the cardinality estimate on which costs depend. Cost estimates might be off by 10 or 15 percent, at most, for a given cardinality, but cardinality estimates can be off by orders of magnitude when their underlying assumptions are invalid or uncertain. Although there has been considerable success in using histograms to detect and correct for data skew [IC91, PIHS96, PI97], and in using sampling to gather up-to-date statistics [HS93, UFA98], there has to date been no comprehensive approach to correcting all modeling errors, regardless of origin.

This paper introduces LEO, the **LE**arning **O**ptimizer, which incorporates an effective and comprehensive technique for a query optimizer actually to learn from any modeling mistake at any point in a QEP, by automatically validating its estimates against actuals for a query after it finishes executing, determining at what point in the plan the significant errors occurred, and adjusting its model dynamically to better optimize future queries. Over time, LEO amasses experiential information that augments and adjusts the database statistics for the part of the database that enjoys the most user activity. Not only does this information enhance the quality of the optimizer's estimates, but it also can suggest where statistics gathering should be concentrated or even can supplant the need for statistics collection. LEO has been prototyped on IBM's DB2 Universal Data Base (UDB) on the Windows, Unix, and OS/2 platforms (hereafter referred to simply as "DB2"), and has proven to be very effective at correcting cardinality estimation errors, as will be demonstrated later.

This paper is organized as follows. Section 2 explores the previous literature in relation to LEO. We give an overview of LEO and an example of its execution in Section 3. Section 4 details how LEO works, including the four major components of capturing the optimizer's plan, monitoring the execution, analyzing the actuals vs. estimates, and exploiting what is learned in the optimizer for subsequent queries. In Section 5, we evaluate LEO's performance – both its overhead and benefit. Section 6 discusses advanced topics and Section 7 contains our conclusions and future work.

# 2 Related Work

Much of the prior literature on cardinality estimates has utilized histograms to summarize the data distribution of columns in stored tables, for use in estimating the selectivity of predicates against those tables. Recent work has extended one-dimensional equi-depth histograms to more sophisticated and accurate versions [PIHS96] and to multiple dimensions [PI97]. This classical work on histograms concentrated on the accuracy of histograms in the presence of skewed data and correlations by scanning the base tables completely, at the price of high run-time cost. The work in [GMP97] deals with the necessity of keeping histograms up-to-date at very low cost. Instead of computing a histogram on the base table, it is incrementally derived and updated from a backing sample of the table, which is always kept up-to-date. Updates of the base table are propagated to the sample and can trigger a partial re-computation of the histogram, but there is no attempt to validate the estimates from these histograms against run-time actuals.

The work of [CR94] and [AC99] are the first to monitor cardinalities in query executions and exploit this information in future compilations. In [CR94] the result cardinalities of simple predicates after the execution of a query are used to adapt the coefficients of a curve-fitting formula. The formula approximates the value distribution of a column instead of employing histograms for selectivity estimates. In [AC99] the authors present a query feedback loop, in which actual cardinalities gleaned from executing a query are used to correct histograms. Multiple predicates can be used to detect correlation and update multi-dimensional histograms. This approach effectively deals with single-table predicates applied while accessing a base table, but the paper does not deal with join predicates, aggregation, and other operators, nor does it specify how the user is supposed to know on which columns multi-dimensional histograms should be created. LEO's approach extends and generalizes this pioneering work. It can learn from any modeling error at any point in a QEP, including errors due to local predicates, expressions of base columns involving user-defined functions, predicates involving parameter markers or host variables, join predicates, keys created by the DISTINCT or GROUP BY clauses, derived tables, and any correlation between any of the above. Most of these operations that change cardinality in some way cannot be addressed by histograms. LEO can even adjust estimates of other parameters such as buffer utilization,

sort heap consumption, I/Os, or the actual running time -- the only real limitation to LEO's approach is the overhead of collecting the actuals for those estimates.

Another research direction focuses on dynamically adjusting a QEP after the execution has begun, by monitoring data statistics during the execution (dynamic optimization). In [KDeW98] the authors introduce a new *statistic collector* operator that is compiled into the plan. The operator collects the row stream cardinality and size and decides whether to continue or to stop the execution and re-optimize the remainder of the plan. Query scrambling in [UFA98] is geared towards the problem of distributed query execution in wide area networks with uncertain data delivery. Here the time-out of a data-shipping site is detected and the remaining data-independent parts of the plan are re-scheduled until the problem is solved. Both solutions deal with dynamic re-optimization of (parts of) a single query, but they do not save and exploit this knowledge for the next query optimization run. LEO is aimed primarily at using information gleaned from one or more query executions to discern trends that will benefit the optimization of future queries. This benefit is not limited to just the same query, because the exact same query is seldom re-executed in modern data warehouses, data marts, and business intelligence applications. Any query with predicates or aggregation on the same column(s) can exploit LEO's learning. LEO does not (yet) address the issue of changing in mid-stream the QEP of a running query, as did [KDeW98] and [UFA98], although it could. Doing this correctly in a real product needs to resolve many hard issues not addressed by that work, such as determining points where such changes produce correct results (i.e., where data is fully materialized, before any results are returned to the user), and reliably predicting the times to re-optimize and execute a new plan so that they can be traded off against the time to complete the original plan.

# 3 A Learning Optimizer

This section gives an overview of LEO's design, a simplified example of how it learns, and some of the practical issues that it must deal with.

## 3.1 An Overview of LEO

LEO exploits empirical results from actual executions of queries to validate the optimizer's model incrementally, deduce what part of the optimizer's model is in error, and compute adjustments to the optimizer's model.

LEO is comprised of four components: a component to save the optimizer's plan, a monitoring component, an analysis component, and a feedback exploitation component. The analysis component is a standalone process that may be run separately from the DB2 server, and even on another system. The remaining three components are modifications to the DB2 server: plans are captured at compile time by an addition to the code generator, monitoring is part of the run-time system, and feedback exploitation is integrated into the optimizer.

The four components can operate independently, but form a consecutive sequence that constitutes a continuous learning mechanism by incrementally capturing plans, monitoring their execution, analyzing the monitor output, and computing adjustments to be used for future query compilations.

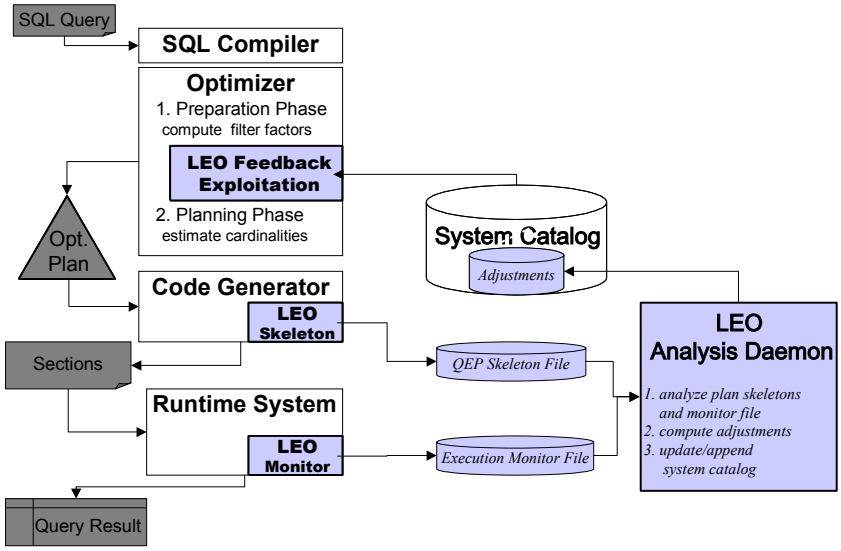



**Figure 1: LEO Architecture**

Figure 1 shows how LEO is integrated into the architecture of DB2. The left part of the figure shows the usual query processing flow with query compilation, QEP generation and optimization, code generation, and code execution. The gray shaded boxes show the changes made to regular query processing to enable LEO's feedback loop: for any query, the code generator dumps essential information about the chosen

QEP (a plan "skeleton") into a special file that is later used by the LEO analysis daemon. In the same way, the runtime system provides monitored information about cardinalities for each operator in the QEP. Analyzing the plan skeletons and the runtime monitoring information, the LEO analysis daemon computes adjustments that are stored in the system catalog. The exploitation component closes the feedback loop by using the adjustments in the system catalog to provide adjustments to the query optimizer's cardinality estimates.

## 3.2 Monitoring and Learning: An Example

In the following we use as an example the SQL query:

```
SELECT * FROM X, Y, Z
WHERE X.Price >= 100 AND Z.City = 'Denver' and Y.Month = 'Dec'
AND X.ID = Y.ID AND Y.NR = Z.NR
GROUP BY A
```

Figure 2 shows the skeleton of a QEP for this statement, including the statistical information and estimates that the optimizer used when building this plan. In addition, the figure also shows the actual cardinalities that the monitoring component of LEO determined during query execution.

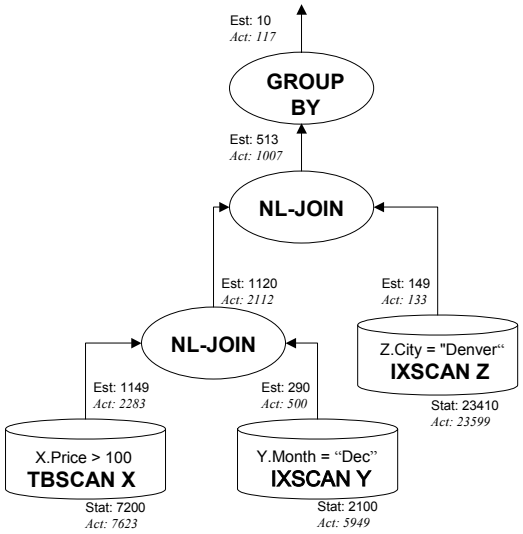



**Figure 2: A QEP Skeleton with Estimates and Actuals**

In the Figure, cylinders indicate base table access operators such as index scan (IXSCAN) or table scan (TBSCAN), ellipses indicate further operators as nested loop joins (NLJOIN) and grouping (GROUP BY). For the base tables X, Y, and Z, "Stat" denotes the base table cardinality as stored in the system catalog statistics. The optimizer uses this cardinality in its cardinality estimation model to compute an

estimate ("Est") for the result cardinality of each table access operator after application of the predicate (e.g., X.Price >= 100) as well as for each of the nested-loop join operators. During query execution, the LEO monitoring component measures the comparable actual cardinality ("*Act*") for each operator.

Comparing actual and estimated cardinalities enables LEO to give feedback to the statistics that were used for obtaining the base table cardinalities, as well as to the cardinality model that was used for computing the estimates. This feedback may be a positive reinforcement, e.g., for the table statistics of Z, where the table access operator returned an actual cardinality for Z that is very close to that stored in the system catalog statistics. The same holds for the output cardinalities of each operator, such as a positive feedback for the estimate of the restriction on Z that also very closely matches the actual number. However, it may also be a negative feedback – as for the table access operator of Y, where the statistics suggest a number almost three times lower than the actual cardinality – or for the join estimates of the nested-loop join between X and Y. In addition, correlations can be detected, if the estimates for the individual predicates are known to be accurate but some combination of them is not. In all of the above, "predicates" can actually be generalized to any operation that changes the cardinality of the result. For example, the creation of keys by a DISTINCT or GROUP BY clause reduces the number of rows. DB2's estimates for this reduction can also be adjusted by LEO, and may also be correlated with predicates applied elsewhere in the query.

All of this feedback is used by LEO to help the optimizer learn to better estimate cardinalities the next time a query involving these tables, predicates, joins, or other operators is issued against the database.

## 3.3 Practical Considerations

In the process of implementing LEO, several practical considerations became evident that prior work had not addressed. We now discuss some of these general considerations, and how they affected LEO's design.

### 3.3.1 The Hippocratic Oath: "Do no harm!"

The overall goal of LEO is to improve query performance by adjusting existing statistics based upon previously executed queries. Ideally, these adjusted statistics provide a better decision basis for selecting the best execution plan for a query. However, this learned knowledge must be arrived at extremely conservatively: LEO should not make hasty conclusions based upon inconclusive or spotty data. And it

must be used carefully: under no circumstances should LEO make things worse! In critical applications, stability and reliability of query processing are often favored over optimality with occasional unpredictable behavior. If adjustments are immediately taken into account for query optimization, even on a highly dynamic database, the same query may generate a different execution plan each time it is issued and thus may result in a thrashing of execution plans. This instability can be avoided if re-optimization of queries takes place after the learned knowledge has converged to a fixed point or by reaching a defined threshold of reliability. Thus a typical usage pattern of LEO might be an initial phase of learning, followed by a stable phase where the execution plans are frozen in order to obtain fast, reliable query processing.

### 3.3.2 Modifying Statistics vs. Adjusting Selectivities

A key design decision is that LEO never updates the original catalog statistics. Instead, it constructs a second set of statistics that will be used to adjust (i.e. repair) the first, original layer. The adjustments are stored as special tables in the system catalog. The compilation of new queries reads these adjustments, as well as the base statistics, and adjusts the optimizer's estimates appropriately. This two-layered approach has several advantages. First, we have the option of disabling learning, by simply ignoring the adjustments. This may be needed for debugging purposes or as a fallback strategy in case the system generated wrong adjustments or the new optimal plan shows undesired side effects. Second, we can store the specific adjustment value with any plan that uses it, so that we know by how much selectivities have already been adjusted and avoid incorrect re-adjustments (no "deltas of deltas"). Lastly, since we keep the adjustments as catalog tables, we introduce an easily accessible mechanism for tuning the selectivities of query predicates that could be updated manually by experienced users, if necessary.

### 3.3.3 Consistency between Statistics

DB2 collects statistics for base tables, columns, indexes, functions, and tablespaces, many of which are mutually interdependent. DB2 allows for incremental generation of statistics and checks inconsistencies for user-updateable statistics. LEO also must ensure the consistency of these interdependent statistics. For example, the number of rows of a table determines the number of disk pages used for storing these rows. When adjusting the number of rows of a table, LEO consequently also has to ensure consistency with the number of pages of that table -- e.g., by adjusting this figure as well -- or else plan choices will be biased.

Similarly, the consistency between index and table statistics has to be preserved: If the cardinality of a column that is (a prefix of) an index key is adjusted in the table statistics, this may also affect the corresponding index statistics.

### 3.3.4 Currency vs. Accuracy

Creating statistics is a costly process, since it requires scanning an entire table or even the entire database. For this reason, database statistics are often not existent or not accurate enough to help the optimizer to pick the best access plan. If statistics are expected to be outdated due to later changes of the database or if no statistics are present, DB2 fabricates statistics from the base parameters of the table (file size from the operating system and individual column sizes). The presence of adjustments and fabricated statistics creates a decision problem for the optimizer -- it must decide whether to believe possibly outdated adjustments and statistics, or fuzzy but current fabricated statistics.

When statistics are updated, many of the adjustments calculated by LEO no longer remain valid. Since the set of adjustments that LEO maintains is not just a subset of the statistics provided by RUNSTATS, removing all adjustments during an update of the statistics might result in a loss of information. Therefore any update of the statistics should re-adjust the adjustments appropriately, in order to not loose information like actual join selectivities and retain consistency with the new statistics.

### 3.3.5 LEO vs. Database Statistics

Note that LEO is not a replacement for statistics, but a rather a complement: LEO gives the most improvement to the modeling of queries that are either repetitive or are similar to earlier queries, i.e., queries for which the optimizer's model exploits the same statistical information. LEO extends the capabilities of the *RUNSTATS* utility by gathering information on derived tables (e.g., the result of several joins) and gathering more detailed information than RUNSTATS might. Over time, the optimizer's estimates will improve most in regions of the database that are queried most (as compared to statistics, which are collected uniformly across the database, to be ready for any possible query). However, for correctly costing previously unanticipated queries, the statistics collected by RUNSTATS are necessary even in the presence of LEO.

# 4 The LEO Feedback Loop

The following sections describe the details of how LEO performs the four steps of capturing the plan for a query and its cardinality estimates, monitoring queries during execution, analyzing the estimates versus the actuals, and the exploitation of the adjustments in the optimization of subsequent queries.

## 4.1 Retaining the Plan and its Estimates

During query compilation in DB2, a code generator component derives an executable program from the optimal QEP. This program, called a *section*, can be executed immediately (dynamic SQL) or stored in the database for later, repetitive execution of the same query (static SQL). The optimal QEP is not retained with the section; only the section is available at run-time. The section contains one or more *threads*, which are sequences of operators that are interpreted at run-time. Some of the section's operators, such as a table access, closely resemble similar operators in the QEP. Others, such as those performing predicate evaluation, are much more detailed. Though in principle it is possible to "reverse engineer" a section to obtain the QEP from which it was derived, in practice that is quite complicated. To facilitate the interpretation of the monitor output for LEO, we chose to save at compile-time a "skeleton" subset of the optimal QEP for each query, as an analysis "road map". This *plan skeleton* is a subset of the much more complete QEP information that may optionally be obtained by a user through an EXPLAIN of the query, and contains only the basic information needed by LEO's analysis, including the cumulative cardinality estimates for each QEP operator, as shown in Figure 2.

## 4.2 Monitoring Query Execution

LEO captures the actual number of rows processed by each operator in the section by carefully instrumenting the section with run-time counters. These counters are incremented each time an operator processes a row, and saved after the query completes. LEO can be most effective if this monitoring is on all the time, analyzing the execution of every query in the workload. For this to be practical, LEO's monitoring component must impose minimal overhead on regular query execution performance. The overhead for incrementing these counters has been measured and shown to be minimal, as discussed in Section 5.1.

## 4.3 Analyzing Actuals and Estimates

The analysis component of LEO may be run off-line as a batch process, perhaps even on a completely separate system, or on-line and incrementally as queries complete execution. The latter provides more responsive feedback to the optimizer, but is harder to engineer correctly. To have minimal impact on query execution performance, the analysis component is designed to be run as a low-priority background process that opportunistically seizes "spare cycles" to perform its work "post mortem". Any mechanism can be used to trigger or continue its execution, preferably an automated scheduler that supervises the workload of the system. Since this means LEO can be interrupted by the scheduler at any point in time, it is designed to analyze and to produce feedback data on a per-query basis. It is not necessary to accumulate the monitored data of a large set of queries to produce feedback results.

To compare the actuals collected by monitoring with the optimizer's estimates for that query, the analysis component of LEO must first find the corresponding plan skeleton for that query. Each plan skeleton is hashed into memory. Then for each entry in the monitor dump file (representing a query execution), it finds the matching skeleton by probing into the skeletons hash table. Once a match is located, LEO needs to map the monitor counters for each section operator back to the appropriate QEP operator in the skeleton. This is not as straightforward as it sounds, because there is not a one-to-one relationship between the section's operators and the QEP's operators. In addition, certain performance-oriented optimizations will bypass operators in the section if possible, thus also bypassing incrementing their counters. LEO must detect and compensate for this.

```
analyze_main(skeleton root) {                                                    (0)
      preprocess (root); error = OK; // construct global state and
                                     // pushdown node properties                  (1)
      for (i = 0; i < children(root); i++) // for each child                     (2)
         {error |= analyze_main(root->child[i]); } // analyze                   (3)
      if (error) return error; // if error in any child: return error            (4)
      switch (root->opcode) // analyze operator                                  (5)
         case IXSCAN: return analyze_ixscan(root)                                (6)
         case TBSCAN: return analyze_tbscan(root)                                (7)
         case …
```

**Figure 3: LEO algorithm**

The analysis of the skeleton tree is a recursive post-order traversal (cf. Figure 3). Before actually descending down the tree, a preprocessing of the node and its immediate children is necessary to

construct global state information and to push down node properties (1). The skeleton is analyzed bottom up, where the analysis of a branch stops after an error occurred in the child (4). Upon returning from all children, the analysis function of the particular operator is called (6, 7, …).

### 4.3.1 Calculating the Adjustments

Each operator type (*TBSCAN, IXSCAN, FETCH, FILTER, GROUP BY, NLJOIN, HSJOIN, etc.*) can carry multiple predicates of different kinds (*start/stop keys, pushed down, join*). According to the processing order of the predicates within the operator, LEO will find the actual monitor data (input and output cardinalities of the data stream for the predicate) and analyze the predicate. By comparing the actual selectivity of the predicate with the estimated selectivity that was stored with the skeleton, LEO deduces an adjustment factor such that the DB2 optimizer can later compute the correct selectivity factor from the old estimate and the new adjustment factor. This adjustment factor is immediately stored in the database in new LEO tables. Note that LEO does not need to re-scan the DB2 catalog tables to get the original statistics, because the estimates that are based on these statistics are stored with the skeleton.

LEO computes an adjustment such that the product of the adjustment factor and the estimated selectivity derived from the DB2 statistics yields the correct selectivity. To achieve that, LEO uses the following variables that were saved in the skeleton or monitor result:

- *old_est*: the estimated selectivity from the optimizer
- *old_adj*: an old adjustment factor that was possibly used to compute old_est
- *act*: The actual selectivity that is computed from the monitor data

After detecting an error ( $| old\_est - act | / act > 0.05$ ) for the predicate $col < X$, LEO computes the adjustment factor so that the new estimate equals the actual value (*act*) computed from the monitor: $est = actual = stats * adj$ ; where stats is the original selectivity as derived from the catalog. The old estimate (*old_est*) is either equivalent to the original statistic estimate (*stats*) or was computed with an old adjustment factor (*old_adj*). Hence this old adjustment factor needs to be factored out. ($adj = act / stats = act/(old\_est/old\_adj) = act*(old\_adj/old\_est)$.

Since the selectivity for the predicate ($col >= X$) is 1 – selectivity($col < X$), we invert the computation of the estimate and the adjustment factor for this type of predicate. Note that we derive an adjustment factor

for the < -operator from the results of the > -operator, and we apply the adjustment factor of a < -operator for the computation of the > -operator. Figure 4 summarizes some of the formulas for computing the adjustments (LEO) and the new estimates (DB2 optimizer).

| PREDICATE | ADJUSTMENT | NEW ESTIMATE |
|---|---|---|
| None, Table Access | adj = act_card * old_adj / old_est | est_card = stats_card * adj |
| Column < Literal<br>Column <= Literal<br>Column = Literal; | adj = act * old_adj / old_est | est = stats*adj |
| Column > Literal<br>Column >= Literal | adj = (1-act)* old_adj/(old_est$_{<}$) | est = 1-est$_{<}$*adj |
| Column = Column | adj = act * old_adj / old_est | est = stats*adj |
| Column LIKE Literal | adj = act * old_adj / old_est | est = stats*adj |
| Complex / UDF | adj = act * old_adj / old_est | est = stats*adj |

**Figure 4: Calculating Adjustments and Estimates**

Using the example from Figure 2 and a TBSCAN on table X with the predicate Price >= 100, we can compute the adjustment factors for the table cardinality and the predicate. The cardinality adjustment factor is 7632/7200 = 1.06. The estimated selectivity of the predicate was 1149/7200 = 0.1595 while the actual selectivity is 2283/7632 = 0.2994. The adjustment factor for the corresponding *Price < 100* -predicate is (1 - 0.2994) * 1.0 / (1 -0.1595) = 0.8335. The optimizer will compute the selectivity for this predicate in the future to be 1 – 0.8335 * (1 – 0.1595) = 0.2994. The adjusted table cardinality of the TBSCAN (1.06*7200) times the adjusted predicate selectivity 0.2994 computes the correct, new estimate of the output cardinality of the TBSCAN operator (2283).

However, different types of section operators can be used to execute a particular predicate such as ‘Price >= 100’. If the Price column is in the index key, the table access method could be an IXSCAN-FETCH combination. If Price is the leading column of the index key, the predicate can be executed as a start/stop key in the IXSCAN operator. Then IXSCAN delivers only those rows (with its row identifier or *RID*) that fulfill the key predicate. FETCH uses each RID to retrieve the row from the base table. If the predicate on Price cannot be applied as a start/stop key, it is executed as *a push-down predicate* on every row returned from the start/stop key search. When using a start/stop key predicate, we scan neither the index nor the base table completely, and hence cannot determine the actual base table cardinality. In order to determine the real selectivity of an index start/stop key predicate, we can only approximate the needed input

cardinality by using the old cardinality estimates, if a previously computed table adjustment factor was used[1] (see the pseudo-code example line (3) and (4) in Figure 5).

```
analyze_ixscan(skeleton IXSCAN) {
       if (start_stop_key_num == 0) // no start stop key:                        (1)
       {   tcard = get_act_table_card(); // we have an actual table cardinality
           rc = compute_table_adjustment(tcard, est_card); //compute/store new adj
           if (pushed_down_pred_num >= 1)                                        (2)
           {
              // get the ouput cardinality of the first pred from monitor
              pcard = push_pred[0]->get_pcard();
              act_sel = pcard/tcard; // this it the actual selectivity
              // compute/store a new adjustment factor
              rc |= push_pred[0]->compute_adj(act_sel);
              if (pushed_down_pred_num > 1)
                 { // deal with the second predicate, possibly detecting
                  // a combined error i.e. a correlation
                 }
           }
           return rc;
       }
       if (start_stop_key_num == 1) // we do not know the table card.
       {
       // if base card estimate includes old_adj factor we assume
       // our estimate is correct and process the predicate
       if (old_adj)                                                              (3)
           {
              // get the ouput cardinality of the first key pred from monitor
              pcard = start_stop_pred[0]->get_stst_pcard();
              act_sel = pcard/est_card; // predicate selectivity                 (4)
              // compute/store a new adjustment factor
              rc |= start_stop_pred[0]->compute_adj(act_sel);
           }
       if (!rc)
           {
        // continue with pushed_down predicates
           ...
           }
       return rc;
}
```

**Figure 5: Analyze IXSCAN**

The merge-join algorithm demonstrates a similar problem that we have named *implicit early out*. Recall that both inputs of the merge join are sorted data streams. Each row will be matched with the other side until a higher-valued row or no row at all is found. Reaching the end of the data stream on one side

[1] The existence of an adjustment factor indicates that we have seen a complete table scan earlier and successfully repaired an older statistic.

immediately stops the algorithm. Thus any remaining rows from the other side will never be asked for, and hence are not seen or counted by the monitor. As a result, any monitor number for merge-join input streams is unreliable unless we have encountered a dam operator such as SORT, TEMP, or GROUP BY, which ensures the complete scan and count of the data stream prior to the merge join.

### 4.3.2 Storing the Adjustments

After the adjustment factors have been computed, they are stored in an extended system catalog. Figure 6 summarizes the new tables that have been introduced into the DB2 system catalog.

| | | |
|---|---|---|
| **Table** | LEO_TABLES | TabelspaceID, TableID, Adj_factor, Cardinality, Timestamp |
| **Column** | LEO_COLUMNS | TabelspaceID, TableID, ColumnID, Adj_factor, Col_Value, Type, TimeStamp |
| **Join** | LEO_JOINS | TabelspaceID, TableID, ColumnID, J_TabelspaceID, J_TableID, J_ColumnID, Adj_factor, TimeStamp |
| **Keys** | LEO_KEYS | KeyString Adj_factor, Col_Value, Type, TimeStamp |
| **Expression** | LEO_EXPRESSION | ExpressionString Adj_factor, Col_Value, Type, TimeStamp |

**Figure 6: New System Catalog Tables**

Take as an example the column adjustment catalog as stored in LEO_COLUMNS. The first three columns uniquely identify a column (i.e. X.Price), while the *Adj_factor* = 0.8335 and *Col_Value* = '100'. *Timestamp* is the compile time of the query and is used to prohibit learning from old knowledge. *Type* indicates the type of entry: 'F' for a frequent value or 'Q' for a quantile adjustment for the corresponding *Col_Value* value. In LEO_JOINS, a join is sufficiently described by two triplets for the two join columns: (tablespaceID, tableID, columnID, J_tablespaceID, J_tableID, J_columnID). This raises the question as to which column should be the primary column for searching. One obvious solution would be to store each join entry twice, i.e. for a join (T.A = S.B) we would store two rows (2, T, A, 2, S, B, ...) and (2, S, B, 2, T, A, ...), but this would double the overhead of maintaining these entries. Introducing a simple rule of (lexicographic) order on the columns' triplets is sufficient to store the adjustment factors only once: the 'smaller' column (2, S, B) is stored with its join partner (2, T, A) and the adjustment factor. A simple index scan with a search key on the "smaller" join column allows us to efficiently update or retrieve the adjustment factor from the database. LEO_KEYS and LEO_EXPRESSION store the key combination or expression as a character string.

## 4.4 Using Learned Knowledge

Before the DB2 Optimizer begins constructing candidate plans, it first retrieves the schema and statistics for each base table referenced in that query from the catalog cache. From these statistics, the optimizer gets the base-table cardinality and computes selectivity factors for each predicate. At this point, if LEARNING is enabled by a control flag, the optimizer will also search the catalog for any adjustment factors that may be relevant to this query, and adjust the base table statistics, predicate selectivities, and other statistics accordingly. How this is done for each type of adjustment is the subject of this section.

### 4.4.1 Base Table Cardinalities

We start first with adjusting the base table cardinalities, since these are basis for all cardinality estimates of plans. As shown in Table 4, the statistic for the base-table's cardinality need only be multiplied by the adjustment factor, if any, for that table.

As discussed earlier, the difficulty comes in maintaining the consistency of this adjusted cardinality with other statistics for that table. The number of pages in the table, NPAGES, is collected during RUNSTATS and is directly used in the cost model as a more accurate measurement for the number of I/O operations during TBSCAN operations than computing it from the table cardinality, the row width, and the page size. As a result, LEO must adjust NPAGES for base tables, as well as the index statistics (the number of leaf and non-leaf pages) accordingly. In addition, the column cardinalities for each column obviously cannot exceed the table cardinality, but increasing the number of rows may or may not increase the cardinality of any column. For example, adding employee rows doesn't change the cardinality of the Sex column, but probably changes the cardinality of the EmployeeID column. Similarly, the consistency between index and table statistics has to be preserved. If a column that is in one or more index keys has its cardinality adjusted in the table statistics, the corresponding index cardinality statistics (FIRSTKEYCARD, FIRST2KEYCARD, …, FULLKEYCARD) must also be adjusted accordingly.

### 4.4.2 Single-Table Predicates

Next, we consider adjustments to the selectivity of a simple, single-table predicate, illustrated by adjusting the column X.Price for the predicate X.Price < 100. Figure 7 shows the actual cumulative distribution for X.Price. Figure 8 shows the column statistics as well as the corresponding adjustments.

The optimizer computes the selectivity for X.Price < 100 from the statistics by cardinality(X < 100) / Maximal_Cardinality = 500/2000 = 0.25. Applying the adjustments results in adjusted_selecticity(X.Price

< 100) = cardinality(X.Price < 100) * adjustment(X.Price < 100) = 0.25 * 2 = 0.5. If there is no exact match in the statistics for the column value (i.e. X.Price < 100), the adjustment factor is computed by linearly interpolating within the interval in which the value '100' is found.[2]

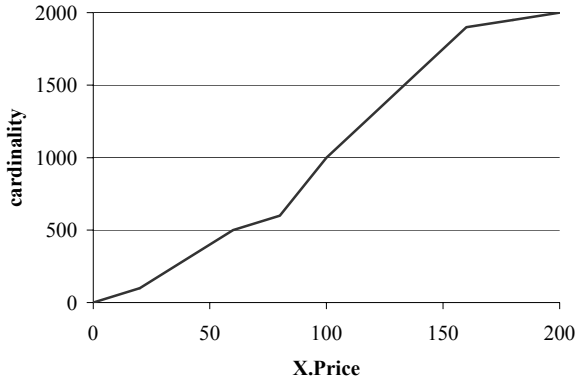



**Figure 7: Actual Data Distribution**

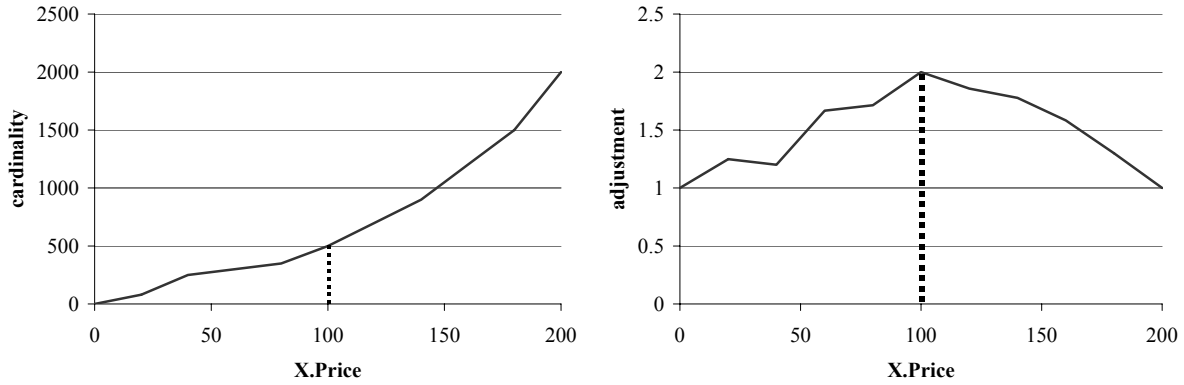



**Figure 8: Column Statistics and Adjustments**

In Figure 9, statistics do not exist (which is equivalent to a default selectivity of 1/3, i.e., a uniformly distributed cardinality of 667). The adjustment curve here shows higher or lower amplitudes than the one for the statistics. For our example: adjustment(X.Price < 100) = 1.5.

Suppose that the optimizer had used an earlier adjustment factor of 2 to compute the estimate for the predicate 'X.Price < 100'. Suppose further that, due to more updates, the real selectivity of the predicate is 0.6 instead of the newly estimated 0.5. The LEO daemon needs to be aware of this older adjustment factor to undo its effects.

[2] The neutral adjustment value of 1.0 is used if LEARNING is disabled or no adjustments (not even using interpolation) are available.

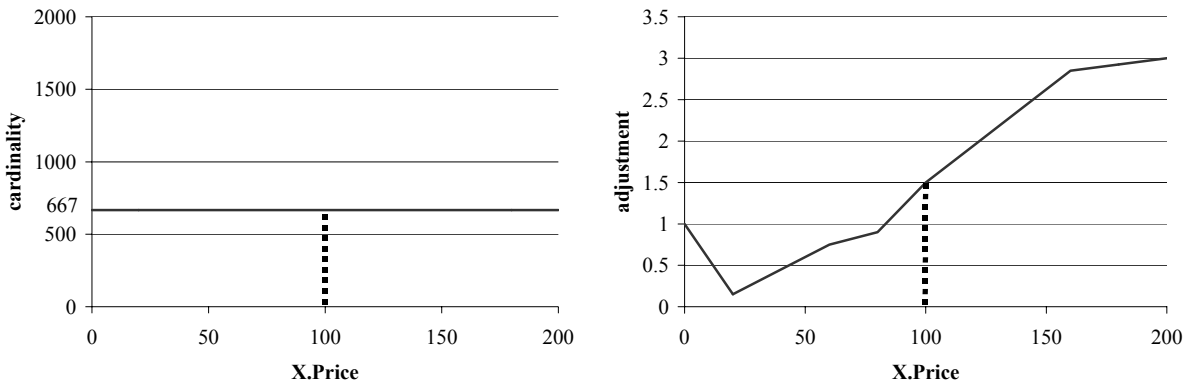



**Figure 9: Adjustments without Statistics**

In our model, an adjustment factor is always based on the systems statistics and never an adjustment of an older adjustment. The new factor is computed by actual_selectivity * old_adjustment / estimate = 0.6 * 2 / 0.5 = 2.4. Thus any previously used adjustment factor must be saved with the QEP skeleton. Note that it is not sufficient to look up the adjustment factor in the system table, since LEO cannot know if it was actually used for that query or if it has changed since the compile time of that query.

The LEO approach is not limited to simple relational predicates on base columns, as is the histogram approach of [AC99]. The "column" could be any expression of columns (perhaps involving arithmetic or string operations), the "type" could be LIKE or user-defined functions, and the literal could even be "unknown", as with parameter markers and host variables. We need only match the predicate's pattern in the LEO_EXPRESSION catalog table and find the appropriate adjustment factor.

### 4.4.3 Join Predicates

As indicated above, LEO can also compute adjustment factors for equality join operators. The adjustment factor is simply multiplied by the optimizer's estimate. Note that having the actuals and estimates for each operator permits LEO to eliminate the effect of any earlier estimation errors in the join's input streams.

### 4.4.4 Other Operators

The GROUP BY and DISTINCT clauses effectively define a key. An upper bound on the resulting cardinality of such operations can be derived from the number of distinct values for the underlying column(s): the COLCARD statistic for individual columns, or the FULLKEYCARD statistic for indexes, if any, on multiple columns. However, predicates applied either before or after these operations may reduce the real cardinalities resulting. Similarly, set operations such as UNION (DISTINCT), UNION

ALL, and EXCEPT may combine two or more sets of rows in ways that are difficult for the optimizer to predict accurately. LEO's analysis routine can readily compute the adjustment factor as adj = act * old_adj / old_est, and adjust the cardinality output by each of these operators by multiplying its estimate by adj. It is doubtful that the histogram approach of [AC99] could provide adjustments for these types of operations in SQL.

### 4.4.5 Correlation between predicates

Optimizers usually assume *independence* of columns. This allows for estimating the selectivity of a conjunctive predicate as a product of the selectivity of the atomic predicates. However, *correlations* sometimes exist between columns, when the columns are not independent. In this case, the independence assumption underestimates the selectivity of a conjunctive predicate.

For example, suppose we have a table storing a computer equipment inventory and request the owners of all IBM Thinkpad T20 notebooks:

```
SELECT owner FROM equipment WHERE supplier = "IBM" AND model = "T20"
```

With 10 suppliers and 100 models, this implies *sel*(supplier = "IBM") = 1/10 and *sel*(model ="T20") = 100.

Without correlation, we obtain

*sel*(supplier = "IBM" and model = "T20") = *sel*(supplier = "IBM") * *sel* (model = "T20") = 1/1000

as the overall selectivity of the query. However, since only IBM produces Thinkpads, we actually have

*sel*(supplier = "IBM" and model = "T20") = *sel*(model = "T20") = 1/100

In practical applications, data is often highly correlated. Types of correlations include functional dependencies between columns and referential integrity, but also more complex cases such as a constraint that a part is supplied by at most 20 suppliers. Furthermore, correlations may involve more than two columns, and hence more than two predicates. Therefore, any set of predicates may have varying degrees of correlation. How are errors due to correlation discerned from errors in the selectivities of the individual predicates? LEO's approach is to first correct individual predicate filter factors, using queries that apply those predicates in isolation. Once these are adjusted, any errors when they are combined must be attributable to correlation. A single query can provide evidence that two or more columns are correlated for specific values; LEO must cautiously mine the execution of multiple queries having

predicates on the same columns before it can safely conclude that the two columns are, in general, correlated to some degree. The multi-dimensional histogram approach of [AC99] could be used here, but presumes that the user knows which columns are correlated and predefines a multi-dimensional histogram for each. LEO can automatically detect good candidates for these multi-dimensional histograms through its analysis.

In our current implementation of LEO, we only take advantage of correlations between join columns. An extension of LEO might take further advantage of correlation in order to provide even better adjustments.

# 5 Performance

## 5.1 Overhead of LEO's Monitoring

LEO requires monitoring query execution in order to obtain the actual cardinalities for each operator of a QEP. Our performance measurements on a 10 GB TPC-H database [TPC00] show that for our prototype of LEO the monitoring overhead is below 5% of the total query execution time, and therefore may be neglected for most applications. Figure 10 shows the actual measurement results for the overhead for TPC-H queries Q2 and Q14, measured both on a single-CPU (serial) and on an SMP machine. These overheads were measured on a LEO prototype. For the product version, further optimizations of the monitoring code will reduce the monitoring overhead even further.

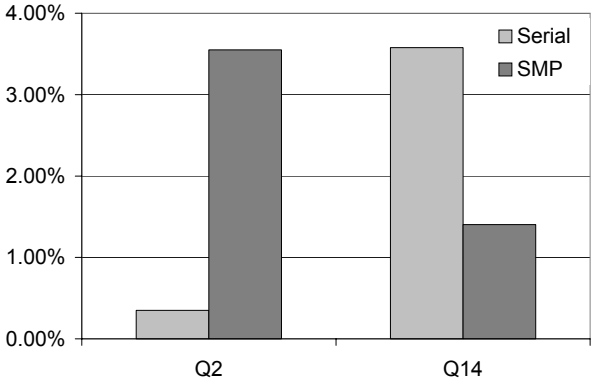



**Figure 10: Monitoring Overhead for a 10 GB TPC-H Database**

Our architecture permits dynamically enabling and disabling monitoring, on a per-query basis. If time-critical applications cannot accept even this small overhead for monitoring, and thus turn monitoring off, they can still benefit from LEO, as long as other – uncritical – applications monitor their query execution and thus provide LEO with sufficient information.

## 5.2 Benefit of Learning

Adjusting outdated or incorrect information may allow the optimizer to choose a better QEP for a given query. Depending on the difference between the new and the old QEP, the benefit of LEO may be a drastic speed-up of query execution

Suppose now that the database in our example has changed significantly since the collection of statistics: the Sales stored in table Y increased drastically in December and the inventory stored in table X received many updates and inserts, where most new items had a price greater than 100. This results in an overall cardinality of more than 21623 records for X and 17949 records for Y. Suppose further that these changes also introduce a skew in the data distribution, changing the selectivities of the predicates X.Price > 100 and Y.Month = 'Dec'. Finally, suppose that a query referencing table X with the predicate X.Price > 150[3], and another query referencing Y with the predicate Y.Month ='Dec', have been executed, providing LEO with some adjustments. Figure 11 shows how LEO changes the query execution plan for the query of Section 3.2 after these changes. The optimizer now chooses to use a bulk method for joining X and Y for this query, thus replacing the nested-loop join with a hash join. Note that the index scan on Y was also replaced by a table scan, due to the adjustments. This new plan resulted in an actual execution speed-up of more than one order of magnitude over the earlier plan executing on the same data.

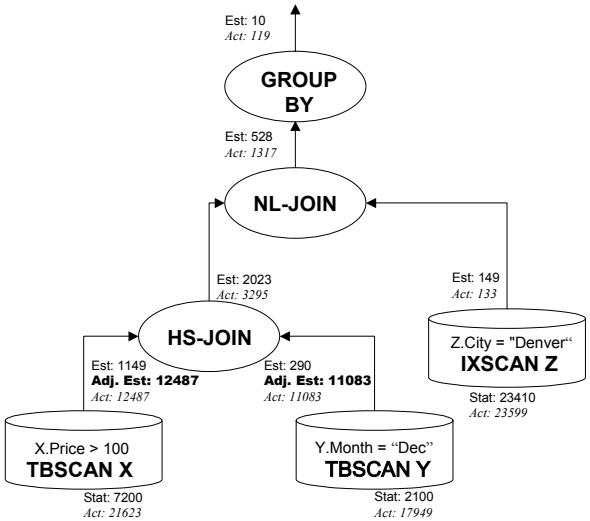



**Figure 11: Execution Plan after Learning LEO's Adjustments**

Our experiments on two highly dynamic test databases (artificial schema and TPC-H) showed that the

[3] Note that it is not necessary to run a query with exactly the predicate X.Price > 100, since LEO performs interpolation for histograms. Thus an adjustment for X.Price >150 is also be useful for a query X.Price > 100.

adjustments provided by LEO enabled the optimizer to choose a QEP that performed up to 14 times better than the QEP without adjustments, while LEO consumed an insignificant runtime overhead, as shown in Section 5.1. Of course, speed-ups can be even more drastic, since LEO's adjustments can cause virtually any physical operator of a QEP to change, and may even alter the structure of the QEP. The most prominent changes are table access operators (IXSCAN, TBSCAN), join method (NLJOIN, HSJOIN, MGJOIN), and changing the join order for multi-way joins.

# 6 Advanced Topics

## 6.1 When to Re-Optimize

A static query is bound to a plan that the optimizer has determined during query compilation. With LEO, the plan for a static query may change over time, since the adjustments might suggest an alternative plan to be better than the plan that is currently used for that query. The same holds for dynamic queries, since DB2 stores the optimized plan for a dynamic query in a statement cache.

Currently we do not support rebinding of static queries or flushing the statement cache because of learned knowledge. It remains future work to investigate whether and when re-optimization of a query should take place. Especially for re-optimization, the Hippocratic Oath must be taken into account, since the actual goal of the statement cache and static queries is to avoid re-optimization. Thus the trade-off between re-optimization and improved runtime must be weighed in order to be sure that re-optimization will result in improved query performance.

## 6.2 Learning Other Information

Learning and adapting to a dynamic environment is not restricted to cardinalities and selectivities. Using a feedback loop, many configuration parameters of a DBMS can be made self-tuning. If, for instance, the DBMS detects by query feedback that a sort operation could not be performed in main memory, the sort heap size could be adjusted in order to avoid external sorting for future sort operations. In the same way, buffer pools for indexes or tables could be increased or decreased according to a previously seen workload. This is especially interesting for resources that are assigned on a per-user basis: Instead of assuming uniformity, buffer pools or sort heaps could be maintained individually per user. If dynamic adaptation is possible even during connections, open but inactive connections could transfer resources to

highly active connections.

Another application of adjustments is to "debug" the cost model of the query optimizer: If – despite correct base statistics – the cost prediction for a query is way off, analyzing the adjustment factors of the plan skeleton permits locating which of the assumptions of the cost model are violated.

Physical parameters such as the network rate, disk access time, or disk transfer rate are usually considered to be constant after an initial set-up. However, monitoring and adjusting the transfer rate for disks and network connection enables the optimizer to act dynamically to the actual workload and use the effective rate.

# 7 Conclusions

LEO provides a general mechanism for the DB2 optimizer to actually learn from its mistakes by adjusting its cardinality and other estimates using the actuals from the execution of previous queries having similar predicates. Regardless of the source of error – old statistics, invalid assumptions, inadequate modeling, unknown literals, etc. – LEO can detect and correct the mistake for any kind of operation that changes the cardinality, at any point in a plan. This is a far more general mechanism than multi-dimensional histograms, which are limited to local predicates on columns of a base table. Our performance measurements have demonstrated that LEO can improve cardinality estimates by orders of magnitude, changing plans to improve performance by orders of magnitude, while adding less than 4% overhead to execution time when monitoring actuals. We feel that LEO provides a major step forward in improving the quality of query optimization and reducing the need for "tuning" of problem queries, a major contributor to cost of ownership.

Our future work includes completing the implementation of LEO's adjustments for all types of predicates, measuring the benefit on a realistic set of user queries, finding conclusive ways to discern correlation among predicates, applying LEO's approach to parameters other than cardinality, and possibly using LEO's adjustments to change a query's plan dynamically during its execution in a robust, industrial-strength way.

# 8 Acknowledgements

The authors thank Kwai Wong for her help with the measurements of the LEO runtime overhead, and Ashutosh Singh and Eric Louie for systems support.

# 9 Bibliography

AC99 A. Aboulnaga and S. Chaudhuri, *Self-tuning Histograms: Building Histograms Without Looking at Data*, SIGMOD Conference 1999

ARM89 R. Ahad, K.V.B. Rao, and D. McLeod, *On Estimating the Cardinality of the Projection of a Database Relation*, ACM Transactions on Databases, Vol. 14, No. 1 (March 1989), pp. 28-40.

CR94 C. M. Chen and N. Roussopoulos, *Adaptive Selectivity Estimation Using Query Feedback*, SIGMOD Conference 1994

Gel93 A. Van Gelder, *Multiple Join Size Estimation by Virtual Domains* (extended abstract), Procs. of ACM PODS Conference, Washington, D.C., May 1993, pp. 180-189.

GMP97 P. B. Gibbon, Y. Matias and V. Poosala, *Fast Incremental Maintenance of Approximate Histograms*, Proceedings of the 23rd Int. Conf. On Very Large Databases, Athens, Greece, 1999

HS93 P. Haas and A. Swami, *Sampling-Based Selectivity Estimation for Joins - Using Augmented Frequent Value Statistics*, IBM Research Report RJ9904, 1993

IBM00 DB2 Universal Data Base V7 Administration Guide, IBM Corp., 2000

IC91 Y.E. Ioannidis and S. Christodoulakis. On the Propagation of Errors in the Size of Join Results, SIGMOD Conference, 1991

KdeW98 N. Kabra and D. DeWitt, *Efficient Mid-Query Re-Optimization of Sub-Optimal Query Execution Plans*, SIGMOD Conference 1998

Lyn88 C. Lynch, *Selectivity Estimation and Query Optimization in Large Databases with Highly Skewed Distributions of Column Values*, Proceedings of the 14th Int. Conf. On Very Large Databases, 1988

PI97 V. Poosala and Y. Ioannidis, *Selectivity Estimation without the attribute value independence assumption*, Proceedings of the 23rd Int. Conf. On Very Large Databases, 1997

PIHS96 V. Poosala, Y. Ioannidis, P. Haas, and E. Shekita, *Improved histograms for selectivity estimation of range predicates,* SIGMOD Conf. 1996, pp. 294-305

SAC+79 P.G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, T. G. Price, *Access Path Selection in a Relational Database Management System*, SIGMOD Conference 1979, pp. 23-34

SS94 A. N. Swami, K. B. Schiefer, *On the Estimation of Join Result Sizes*, EDBT 1994, pp. 287-300

TPC00 Transaction Processing Council, *TPC-H Rev. 1.2.1 specification*, http://www.tpc.org/benchmark_specifications/Tpc-h/h121.pdf , 2000

UFA98 T. Urhan, M.J. Franklin and L. Amsaleg, *Cost-based Query Scrambling for Initial Delays*, SIGMOD Conference 1998

# Current Issues and Solutions

## SRIRAM: A scalable resilient autonomic mesh

Mirroring and replication are common techniques for ensuring fault-tolerance and resiliency of client/server applications. Because such mirroring and replication procedures are not usually automated, they tend to be cumbersome. In this paper, we present an architecture in which the identification of sites for replicated servers, and the generation of replicas, are both automated. The design is based on a self-configuring mesh of computers and a communication mechanism between nodes that operates on a rooted spanning tree. A query-search component uses Java™ language-based query capsules traveling along the branches of the spanning tree, and a caching scheme whereby the query and previous search results are cached at each node for improved efficiency. Furthermore, a security and anonymity component relies on one or more authentication servers and an anonymous communication scheme using link local addresses and indirect communication between the nodes via the spanning tree. The architecture also includes components for resource advertising and for application replication.

☞ **See the Appendix Ⅰ**

## Autonomic service deployment in networks

Networks have been growing dramatically in size and functionality in past years. Internet Protocol network nodes not only forward datagrams using longest-prefix matching of the destination address, but also execute functions based on dynamic policies such as proxy-caching, encryption, tunneling, and firewalling. More recently, programmable behaviors have begun to appear in network elements, allowing experimentation with even more sophisticated services. This paper presents an autonomic approach to network service deployment that scales to large heterogeneous networks. Topological categories of service deployment are introduced. A two-phase deployment mechanism that is split into hierarchically distributed and central computations is presented and illustrated with examples of actual services in a programmable network environment, together with their deployment algorithms and simulation results. Autonomic service deployment allows the distributed and complex capabilities present in network elements to be leveraged more efficiently when installing new services than is possible in traditional centralized network management-based approaches. As a result, installation is faster and use of functional resources is more optimized.

☞ **See the Appendix Ⅱ**

## Enabling autonomic behavior in systems software with hot swapping

Autonomic computing systems are designed to be self-diagnosing and self-healing, such that they detect performance and correctness problems, identify their causes, and react accordingly. These abilities can improve performance, availability, and security, while simultaneously reducing the effort and skills required of system administrators. One way that systems can support these abilities is by allowing monitoring code, diagnostic code, and function implementations to be dynamically inserted and removed in live systems. This “hot swapping” avoids the requisite prescience and additional complexity inherent in creating systems that

have all possible configurations built in ahead of time. For already-complex pieces of code such as operating systems, hot swapping provides a simpler, higher-performance, and more maintainable method of achieving autonomic behavior. In this paper, we discuss hot swapping as a technique for enabling autonomic computing in systems software. First, we discuss its advantages and describe the required system structure. Next, we describe K42, a research operating system that explicitly supports interposition and replacement of active operating system code. Last, we describe the infrastructure of K42 for hot swapping and several instances of its use demonstrating autonomic behavior.

☞ **See the Appendix Ⅲ**

## Toward a new landscape of systems management in an autonomic computing environment

In this paper we present IBM Tivoli Monitoring, a systems management application that displays autonomic behavior at run time, and we focus on extending it in order to encompass the design and the deployment phases of the product life cycle. We review the resource model concept, illustrate it with an example, and discuss its role throughout the product life cycle. Then we introduce basic concepts in ontology and description logics and discuss representing Common Information Model constructs using description logics. Finally, we propose Systems Management Ontology, an approach to enhancing the autonomic properties of IBM Tivoli Monitoring based on an ontology service and the technique of “contextual pulling” applied to the resource model.

☞ **See the Appendix Ⅳ**

## Competitive algorithms for the dynamic selection of component implementations

As component-based development matures, more and more applications are built by integrating multiple distributed components. We suggest providing components with multiple implementations, each optimized for a particular workload, and augmenting the component run-time environment with a mechanism for switching between implementations. This mechanism monitors the types of requests the component is receiving, and adaptively switches implementations for optimal application performance. Achieving this optimal performance depends on making good choices as to when and how to switch implementations, a problem we refer to as the *adaptive component problem*. We first formalize the generic problem and then provide an algorithm, named Delta, for switching implementations in the special case when the component has exactly two implementations. We show that this algorithm is (3 _ epsilon)-competitive with respect to the optimal algorithm, where epsilon is a small fraction. We establish a 3-competitive lower bound for the problem, which implies that Delta is close to optimal. We describe the application of these results to the distributed pub/sub problem, and the data structure selection problem.

☞ **See the Appendix Ⅴ**

## Security in an autonomic computing environment

System and network security are vital parts of any autonomic computing solution. The ability of a system to react consistently and correctly to situations ranging from benign but unusual events to outright attacks is key to the achievement of the goals of self-protection, self-healing, and self-optimization. Because they are often built around the interconnection of elements from different administrative domains, autonomic systems raise additional security challenges, including the establishment of a trustworthy system identity, automatically handling changes in system configuration and interconnections, and greatly increased configuration complexity. On the other hand, the techniques of autonomic computing offer the promise of making systems more secure, by effectively and automatically enforcing high-level security policies. In this paper, we discuss these and other security and privacy challenges posed by autonomic systems and provide some recommendations for how these challenges may be met.

☞ **See the Appendix Ⅵ**

## A system model for dynamically reconfigurable software

The ability to reconfigure software is useful for a variety of reasons, including adapting applications to changing environments, performing on-line software upgrades, and extending base application functionality with additional nonfunctional services. Reconfiguring distributed applications, however, can be difficult in practice because of the dependencies that exist among the processes in the system. This paper formally describes a model for capturing the structure and run-time behavior of a distributed system. The structure is defined by a set of elements containing the state variables in the system. The run-time behavior is defined by threads that execute atomic actions called operations. Operations invoke code blocks to bring about state changes in the system, and these state changes are confined to a single element and thread. By creating input/output signatures based upon the variable access patterns of the code blocks, dataflow dependencies among operations can be derived for a given configuration of the system. Proposed reconfigurations can be evaluated through off-line tests using the formal model to determine whether the new mapping of operations-to-code blocks disrupts existing dataflow dependencies in the system. System administrators—or software components that control adaptivity in autonomic systems—can use the results of these tests to gauge the impact of a proposed reconfiguration on the existing system. The system model presented in this paper underpins the design of reconfigurable ARMOR (Adaptive Reconfigurable Mobile Objects of Reliability) processes that provide flexible error detection and recovery services to user applications.

☞ **See the Appendix Ⅶ**

# SRIRAM: A scalable resilient autonomic mesh

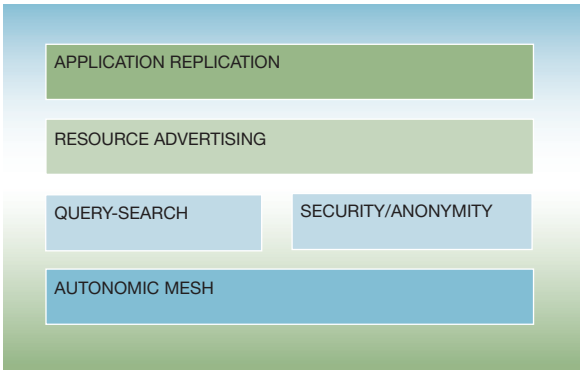
by D. C. Verma
S. Sahu
S. Calo
A. Shaikh
I. Chang
A. Acharya

Mirroring and replication are common techniques for ensuring fault-tolerance and resiliency of client/server applications. Because such mirroring and replication procedures are not usually automated, they tend to be cumbersome. In this paper, we present an architecture in which the identification of sites for replicated servers, and the generation of replicas, are both automated. The design is based on a self-configuring mesh of computers and a communication mechanism between nodes that operates on a rooted spanning tree. A query-search component uses Java™ language-based query capsules traveling along the branches of the spanning tree, and a caching scheme whereby the query and previous search results are cached at each node for improved efficiency. Furthermore, a security and anonymity component relies on one or more authentication servers and an anonymous communication scheme using link local addresses and indirect communication between the nodes via the spanning tree. The architecture also includes components for resource advertising and for application replication.

Most networked applications are currently implemented using a client/server computing model. A server with a well-known address hosts the application, while different clients access it over the network. Typically, the server (or a set of servers) will be located at a single site, and the overall performance of the application will depend upon factors such as the speed of the network between the client and the server, the computing power at the hosting site, and congestion in the network. The concentration of servers at a single site also reduces the ability of the application to withstand failures. In order to improve the availability and reliability of a system, distributed architectures incorporating replicas and mirrors are frequently used. However, the process of replication and mirroring is usually manual and, due to the complexity in the control and management of the system, somewhat cumbersome. An autonomic replication and mirroring facility would significantly simplify the process of replication and would improve the availability of applications.

In this paper, we describe the highly scalable distributed architecture SRIRAM (Scalable Replication Infrastructure using Resilient Autonomic Meshes), which is designed to dynamically create replicas of applications for resilient operation. The basic idea behind SRIRAM is that several computers are available at any given time on the network, and an application deployed on one of the machines can be mirrored and run on any other machine that is available and capable of providing the same service. All the computers are connected in a mesh with self-managing properties. A machine hosting an application uses the communications overlay (an application-level communication network that overlays the mesh) to transmit the application's replication

Figure 1 The SRIRAM architecture




requirements, identify potential replicas, and configure the replicas to start a copy of the application. Clients search for one of the replicas of any application in which they are interested, and invoke the services of that application from the replicated copy.

The autonomic replication infrastructure provided by SRIRAM can be used in various scenarios, but is most relevant in the context of peer-to-peer networks,[1,2] content distribution networks,[3,4] and Grid computing.[5] The basic SRIRAM architectural components can be used to improve the underlying communication infrastructure (resiliency to faults, increased availability, etc.) in each of these contexts, with the application replication mechanism as a specific feature provided within each of these operating environments.

The rest of the paper is structured as follows. In the next section we give an overview of the SRIRAM architecture. In the following four sections we discuss each of the major components of the architecture as well as the types of applications that can exploit the replication support provided by SRIRAM. In the remaining two sections we review related work, and then we present our conclusions and directions for future research.

## Architecture overview

The SRIRAM architecture, which includes five major components, is illustrated in Figure 1. The mesh creates a network interconnecting all machines participating in the system. A flexible and efficient query-search mechanism is built on top of the network. Security and anonymity controls round up the communication infrastructure consisting of the bottom three components. The upper two layers are a specific use of this infrastructure. The query-search mechanism facilitates the resource advertising by the participants on the mesh. The resource advertising facility is used for automatic search and for creation of replicas.

The *autonomic mesh* component consists of a self-configuring network interconnecting all machines in the system. This layer supports a broadcast mechanism that allows all the participating machines to communicate in an efficient, scalable, self-configuring, and self-healing manner.

The *query-search* component supports basic search primitives that allow participants to search for information about other participants within the SRIRAM system. SRIRAM uses a system based on active programs (Java** language-based query capsules), which enables a flexible and efficient search mechanism. Caching is used to improve the responsiveness of the system.

The *security/anonymity* component provides for communication with other peers while preserving the anonymity of the requester or the respondent. Security and access control within SRIRAM is based on digital certificates issued by trusted authentication servers. The *resource advertising* component allows a participating machine to describe the resources required for replicating the applications running on it, and for possible replicas to indicate their resource availability.

Finally, the *application replication* component provides the basic functions for replicating the code and data of applications, and for maintaining the proper consistency of application data among the different mirrors. Each one of these components is described in more detail in subsequent sections.

## Autonomic mesh algorithm

The autonomic mesh component within SRIRAM provides an overlay that interconnects all of the participating machines so that they may communicate with each other. This function is similar to the overlays created in distributed peer-to-peer networks like Gnutella[2] that enable group communication among all participants. However, Gnutella and similar systems use a flooding scheme for their group communication, which consumes a significant amount of network and node resources. In SRIRAM, we have opted

for a scheme that builds a rooted spanning tree among all the participants and attempts to minimize the number of messages exchanged for any given query.

The use of a rooted spanning tree has its own set of problems. The traditional distributed algorithms for creating a spanning tree are relatively slow and complex, and are thus impractical for our needs. Furthermore, nodes that are nearer the root of the rooted spanning tree are likely to see more traffic than nodes at the leaves of the tree. The tree is also more likely to be disrupted when a machine leaves or joins the system.

To accelerate the process of spanning tree creation, SRIRAM uses a semi-distributed scheme similar to that used in YOID.[6] In the semi-distributed scheme, SRIRAM deploys a number of hint-servers within the system. The hint-servers store a limited amount of information about the participants, and the information is not guaranteed to be up-to-date. A participant wishing to join the system communicates with the hint-servers in order to obtain the identities of possible nodes in the existing spanning tree to which it can connect.

To solve the problem of increased load on participants near the root of the tree, a ranking scheme is used. Each participating node in SRIRAM computes a rank for itself. A rank is a measure of the computing capability of the node. For computational ease, the closer the node is to the center of spanning tree activity, the lower its rank. The root of the spanning tree is the node with the lowest rank. In addition, the rank computation also involves the inverse of a weighted combination of its CPU speed, available disk space, memory size, and speed of its network interfaces. Ranks impose a strict ordering on the participants in the tree, and ensure that no cycles can form in the constructed tree.

For efficient tree creation and restructuring, a new machine is only allowed to join the tree by choosing a parent from among existing participants with ranks lower than itself. This provides a simple, yet effective, scheme for eliminating cycles in the spanning tree. The selected participant becomes the parent of the new node. When a participant leaves the tree, its children join the parent of the departing machine. If an orphaned child node does not succeed in joining any node, it then increases its own rank and contacts the hint-server for a list of possible parents. The steps in the creation of the spanning tree are described below in further detail.

**Joining the tree.** When a new machine is about to join a tree (this is known as the registration phase), it computes its rank and contacts the hint-server to obtain a list of machines with ranks slightly lower than the computed rank. It then contacts each of the machines in the list and requests that it become its parent. A machine in the list may accept the request only if it has a lower rank than the newcomer. In addition, it may refuse to accept new children beyond a certain preconfigured limit, or it may no longer be up. The delay in obtaining a response from the machine is used to estimate the round-trip delay between the potential parent and the newcomer. The newcomer joins (as child) the machine with the lowest latency that responds positively to its join request. If no machine in the list responds positively, the newcomer doubles its computed rank and obtains a new list from the hint-server. If a newcomer has a rank smaller than the current root, the hint-server returns a special code to both the current root machine and the newcomer, asking that the newcomer become the new root of the spanning tree as the parent of the existing root machine.

**Tree improvement algorithm.** The node that a newcomer initially selects as its parent may not be the best choice for the system. In order to continually improve the structure of the spanning tree, each node periodically obtains a list of its siblings (the other children of its parent) and the name of its grandparent (the parent node of its parent node). It then assesses the latency and the feasibility of these machines to become its parent. If a machine with lower rank and latency is found, then the node switches over to the new parent. The tree improvement process is an ongoing procedure that tries to optimize the spanning tree configuration, which—given the dynamic arrival and departure of nodes—could otherwise deteriorate over time.

**Data structures at the hint-servers.** Each hint-server in SRIRAM maintains a data structure containing a partial list of the current participants in the system. The participants are maintained in a fixed-size list, sorted according to their ranks. A participant is first entered into the list when, as a newcomer, it queries the hint-server for joining the tree. During the registration phase, each participant indicates the number of children it is able to support. If the number of children is nonzero, the participant is entered into the list. If the capacity of the list is exceeded, the

participant with the highest rank is removed from the list. At the time of a join request, a set of K participants is randomly selected from the next 2K participants with rank higher than the newcomer and returned to the newcomer. Here K is a configuration parameter for the hint-server, with a default value set to be the smaller of 10 and one-hundredth of the number of participants in the system. After a participant's name has been given out more than 2K times, it is removed from the data structure.

The hint-server does not keep track of the nodes' status as they join or leave the system. Therefore, the information provided by the hint-server may be out-of-date, and the participants use the schemes described previously to work around such inaccurate information. Since there is no need to maintain consistent information, a single hint-server can easily support thousands of participants with the only constraint being the amount of storage set aside for its data structures.

**Handling tree partitioning.** Partitioning of the tree is handled by a relatively simple scheme. Each node maintains the identity of the root of the tree to which it belongs. Each root node periodically sends out a message broadcasting its identity along the branches of the tree. The identity of the root is compared in messages exchanged to monitor response times in the tree improvement algorithm. When a node detects that another node in the system has a different notion of the identity of the root node, a root conflict resolution message is sent up to the parents of each node. The root conflict resolution continues up to the roots of the two trees, and the two roots join together with the higher ranked root becoming a child of the lower ranked root. The ranking criteria used by SRIRAM have the effect of establishing well-connected, more powerful nodes as the root node. The root conflict resolution messages are expected to be generated relatively infrequently.

**Self-configuration of the autonomic mesh.** For proper operation of the autonomic mesh, each participant node needs to obtain values for a number of configuration parameters. Examples of such parameters are the frequency at which each node probes its neighbors for tree climbing, the weights used to compute the rank of a participant, and the maximum lifetime of a message sent on the mesh. Other components of the system described later in the paper also need specific configuration parameters, for example, the types of query-capsules that are defined within the system, the identity of certificate servers, and so on. Although each node could choose its own value for a configuration parameter, this would make the entire process more complicated and more difficult to manage.

In order to automate the configuring process, it is assumed that the configuration parameters of an operational SRIRAM system are managed at a central point by an "operator" of the system. The operator maintains a copy of the configuration at the hint-server, and signs it using its public key. The public key and the identity of the operator are available in the digital certificate of the operator. Each participant can obtain a copy of the configuration from the hint-server. The owner of each participant is free to modify the values of its parameters. Alternatively, when a participant attaches, as child, to a new neighbor on the spanning tree, it can obtain the configuration information from the neighbor, validate the signature of the operator, and then use the configuration. The configuration distribution process makes the participating node largely self-configuring (except for those participants who wish to override the operator-specified configuration).

## Query-search mechanism

A client node that wishes to locate a resource, whether a file or a service, sends a query along the spanning tree. The query is encapsulated in a query capsule, which is a piece of Java code that, when executed, will match the search criteria provided by the client node against the resources located at the node on which it is being executed. This query capsule is propagated by each node along all the branches of the spanning tree (except the one that sent the query) and executed at each node it traverses. When a node finds a match for the query, it sends a positive response containing its location back along the spanning tree toward the client node. Each intermediate node that receives this positive response caches both the query and the location of the resource so subsequent searches for the same resource will receive a speedier response. If an intermediate node receives multiple positive responses, it may choose to cache some number of them and return the list in response to subsequent queries.

This concept of query capsules is borrowed from active network schemes and permits the formulation of generic queries. Unfortunately, allowing query capsules to execute on nodes presents both security and performance issues. SRIRAM handles this by providing a set of standard query capsules and by al-

lowing each node to restrict the execution of other arbitrary query capsules. The standard query capsules can contain both simple query capsules and complex query capsules. A simple query capsule may search the contents of a file looking for a key word match, whereas a complex query capsule may use more sophisticated techniques to search for resources that provide a specific Web service, for example. It is assumed that all nodes will permit execution of the standard query capsules, so a client node wishing to conduct a search using one of these standard query capsules need only provide the parameters to that capsule in its search query.

Once a positive response has been cached at an intermediate node, any subsequent queries arriving at the node will first be checked against the cache. If the query capsule appears in the cache, or the query can be answered using the results of a query found in the cache, then this node will contact the node listed in the cache as the location of the resource, in order to ensure the information is still valid. If the cache entry is still valid, the intermediate node will send a positive response containing the location of the resource to the client that originated the search and stop further queries from flooding the spanning tree. For popular search topics, caching can thus considerably reduce the number of search messages and save both network usage and query capsule executions at various nodes.

When a client receives a positive response to its query, it may contact the node offering the resource either directly or indirectly to obtain the needed files or invoke the desired service. When the resource provider node is contacted directly, the client may be asked to authenticate itself by providing a certificate issued by the authentication server. Anonymity on queries is also supported, as described in the section "Security and anonymity."

For a simple example of a query search, see Figure 2. $N_1$ through $N_7$ are nodes in the spanning tree. The client node ($N_1$) sends a query capsule up the tree (see path Q) and this query capsule is executed at each intermediate node until node $N_5$ finds a match. Node $N_5$ generates a positive response containing its location and sends it back along the tree toward the client node $N_1$ (see path R). At each intermediate node ($N_3$ and $N_2$) the query capsule and resource location are cached before being forwarded. Once the client node receives this positive response to its search, it is free to directly (or indirectly) contact node $N_5$ and request the desired resources.

Figure 2 Query-search example

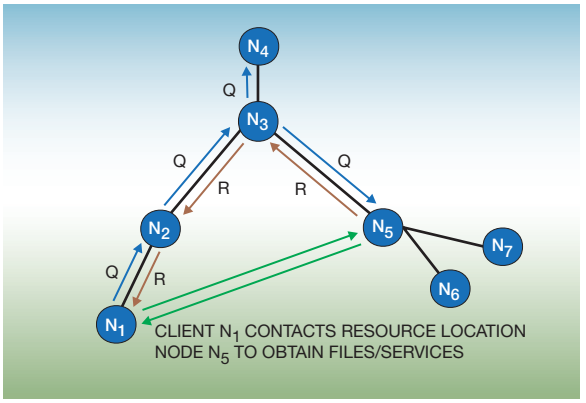
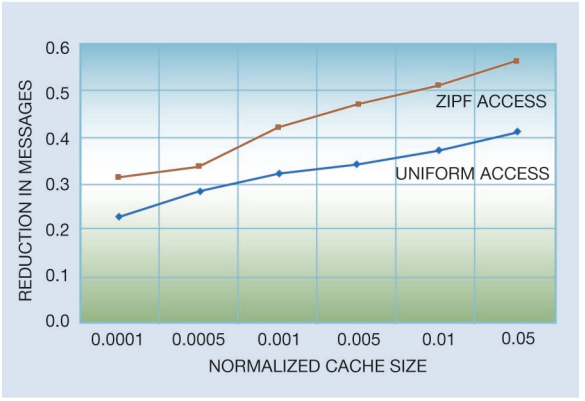



Figure 3 Benefit of query caching




The dynamic improvement of the spanning tree can result in the creation of temporary cycles in the system. In order to eliminate endless looping of queries, each query capsule starts with a preset time-to-live counter, which is decremented every time the query is forwarded by a participant, with the query being discarded once the time-to-live counter reaches zero.

In Figure 3 we present preliminary simulation results illustrating the benefits of caching query results at intermediate nodes. There are two key questions that must be answered in caching the queries: which nodes should cache the queries, and what replacement policy should control the cache mechanism. A simple LRU (least recently used) scheme has been assumed as the cache replacement policy. We have

considered the following three choices for addressing the first issue: (1) all the intermediate nodes that receive the query result cache the result, (2) only the end point nodes cache the query result, and (3) an intermediate node that receives the query result caches it with a probability p, $0 \leq p \leq 1$. We consider a system with 500 nodes for this illustration and each node is randomly assigned a rank between 0 and 500. We assume that there are one million documents available in the system. We vary the cache size so that its capacity varies from 100 to 50000 query results.

Figure 3 illustrates the benefit of caching queries as the normalized cache size varies from 0.0001 to 0.05. There are two graphs, representing two different document popularity distributions. The normalized cache size is obtained by dividing the cache capacity by the total number of unique documents in the system. On the *y* axis, we have plotted the reduction in the average number of messages required for satisfying a query normalized by the number of messages required without caching. We observe that there is an appreciable reduction in the average number of messages required to locate an object with query result caching, for both Uniform and Zipf (with Zipf parameter = 0.9) document access popularity. We observe that the benefit is higher when the access popularity follows a Zipf distribution. In this evaluation, we have assumed that the cached results are always consistent. However, in a dynamic scenario, in which nodes could leave or join at any time, cached results may not always be consistent. We are currently evaluating how to address this problem.

## Security and anonymity

In any distributed system, issues related to security and privacy arise. When participants in a mesh are looking for resources, or advertising the availability of resources and services, it may be desirable to maintain their anonymity, or provide that information to a selected set of participants. SRIRAM uses a certificate-based system[7] in order to support privacy and anonymity in its communications.

SRIRAM supports one or more authentication servers. The authentication servers validate the credentials of a participant and issue to the participant a digital certificate. The certificate, signed with the public key of the authentication server, includes the identity of the participant. A separate certificate identifies the groups to which a participant belongs. The certificates are used as part of the challenge-response system to authenticate participants, following the same schemes used by TLS[8] or IPSEC[9] authentication.

When anonymity is desired, the participants need to hide their IP (Internet Protocol) addresses from other participants. SRIRAM uses a scheme based on link local addresses borrowed from the concept of automatic network routing (ANR) proposed in some broadband communication systems.[10] Each participant assigns a random address to its children and parent. The mapping of the address to the real neighbor in the link is only known to the local node.

Anonymous communication is always indirect, using the spanning tree, rather than direct between the involved parties. Two chains of link local addresses are included in anonymous communication, each chain encoding an anonymous path from one sender to the other. The only exceptions are anonymous queries on the spanning tree, which are used to discover the initial chain of link local addresses to use.

A participant anonymously looking for available resources will send out a query on the spanning tree. Each participant will include the link local address of the neighbor from which it has received the query before forwarding it on to the other branches of the spanning tree. The link local addresses are appended to a growing chain of the path to the requester and form the anonymous path back to the querying participant. When a participant sends a response to a query, it removes the last link local address from the local chain, and sends the message to the neighbor with the specified link local address. These responses are also subject to the reverse path accumulation, and create a reverse path to the respondent.

The anonymous communication process is best illustrated by an example. Consider the system of six nodes shown in Figure 4. Each node assigns link local addresses to members on the spanning tree as indicated by labels in the figure; for example, node B has assigned label 4 to its neighbor A, label 7 to its neighbor C, and label 9 to its neighbor E. Let us consider the case of node A sending out an anonymous query on the spanning tree. It sends the query to node B, which creates a link-local chain beginning with 4 (label assigned to A), and forwards it to its other two neighbors C and E. C appends the local link label of B to the chain, which now becomes 45, and forwards the query to D and F. D appends the local link label of C, and has the accumulated path to the sender of 453.

Assuming that D responds to the sender, it uses the last index 3 to send its response back to B, it strips the index from the end of the path before sending it to C. C notes that the response has come from link local D, creates a reverse path of 1 and uses the remaining path of 45 to propagate the response. The last link local address of 5 indicates that the response should go to B. B uses the remaining path of 4 to send message back to A, and appends 7 to the path to the respondent (which is now 17). A is the final recipient, and knows the path to the respondent (171) without knowing that it is D who responded. For further communication, A can use the path 171 to communicate with D, and D can use the path 453 to communicate with A, each being unaware of the other's identity.

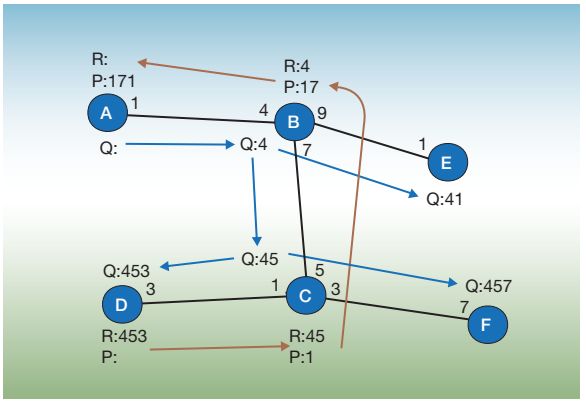
In order for the anonymous communication to work, we only need at least one of the intermediary nodes to play by the rules and not reveal its mapping of link local addresses to others. Each node is aware only of the identity of its immediate neighbors, and is not able to infer the identity of any other participant unless all of the members of the spanning tree along its path collude with it. As the number of participants increases, the ability of any individual to obtain such colluding members becomes negligible.

## Resource advertising and application replication

Before a SRIRAM node runs an application (e.g., an instance of a transcoding service), a standardized description of the application to be run should be provided. The description includes the application type (Web server, directory server, Web service and its description) as well as information required for the implementation of the service. That is, a listing of the code and data that are required to host that application is also included as part of the service description, as well as the configuration required for running that application. The service description allows the copying of the desired software and data components, and the launching of another instance of the application to be automated. The service description also includes the scripts, running at the requesting node, that stop and start the application at the machine providing the service.

In addition to the software configuration, the description also includes values for the minimum amount of disk space, the CPU processing power, the network bandwidth, and any constraints on the operating system needed to run that application. The resource

Figure 4 Anonymous query search




advertising module sends out a query capsule on the spanning tree searching for nodes that may be willing to host a replica of the application. The query capsule may be sent anonymously or with credentials, as specified by the configuration file.

When a participant receives the query capsule, its resource advertisement module examines the locally available system resources. If the available system resources are suitable for running a replica, and if the machine administrator allows running of replicas of other applications, then the resource advertising module sends a response back to the requester indicating the resources available. The response is sent over the spanning tree for anonymous queries and directly to the requester for nonanonymous queries. The IP address of the respondent is included in this type of response.

The requesting node selects a fixed number of replicas from all the responding participants. The heuristic used gives preference to machines with the largest weighted combination of available resources and the largest absolute numeric difference in IP addresses. The application replication module is then invoked to create a replica of the application.

The replicas are created by copying the contents of the software and configuration files and by starting the application using the specified script. SRIRAM allows the configuring of the replication to operate in one of two modes.

- Concurrent execution: An instance of the application is launched on all the new replicas.

- Standby execution: New replicas receive a copy of the code, data, and software needed by the machine. The replicas exchange periodic keep-alive messages with the original node. An instance of the application is only launched on the replica when the original node goes down.

Furthermore, if a data-synchronization script is specified in the standard configuration, the script is executed by each of the replicas in order to obtain the latest data changes from the original application node in both of these modes.

If a node with replicas in the system leaves and then rejoins the network, it searches for the existing replicas in operation by floating a query on the spanning tree. The replicas that it discovers cooperate with the node in order to synchronize the code and data. The synchronization process can also be performed at periodic intervals, as determined by the originating node.

The SRIRAM architecture can be used to support automated replication of many different types of applications. Following are some of the common ones.

- *File and video servers*: It is a common practice to use multiple mirrored sites for providing scalable and resilient HTTP (HyperText Transfer Protocol) service, FTP (file transfer protocol) service, and other services with relatively static content. An instance of an application running on a machine, along with its associated data content, can automatically be replicated to other nodes in SRIRAM.
- *Web services*: Web services[11] use the common paradigm of exporting a WSDL (Web Services Description Language) interface for the operations that they support. For most common Web services, a description of the code components (Java classes, beans, etc.) needed for running the Web service is also required. Such a description, and specific software requirements (e.g., a Tomcat server,[12] or the need for JDK** 1.3.1 operating environment) can be advertised and replicated. The presence of replicas can be documented by the replicas themselves in the UDDI directory, which provides a catalog of services.
- *Database applications*: Applications that make heavy use of dynamically changing data in large databases are hard to replicate due to the overheads associated with synchronizing distributed state. For such applications, the replication process would primarily consist of creating hot standbys that can take over in the case of primary system failure. The application software and database replication process can be created automatically on the replicated sites. The replicas will only become operational in the case of failure.

## Related work

Earlier work on overlay broadcast and multicast architectures covered a number of approaches, including centralized directory servers,[13–15] flooding-based solutions[2,16] that are typically inefficient, slow distributed spanning tree formation,[17] or requiring voluminous state information in each node.[18] Other work has focused on efficient lookup mechanisms.[19–21] These, however, require exact identifiers for their lookup algorithms, which cannot handle the rich queries (e.g., queries using wildcards) desired. Work on application-layer multicast (e.g., References 6, 22, 23) was primarily directed toward building and maintaining efficient overlay meshes, without considerations of application replication and anonymity. The spanning tree algorithm in Reference 6 does not account for the network and other resources available at each node in the tree construction. While the approach in Reference 22 improves the tree construction process of Reference 6 by first considering a mesh formation and then constructing the tree, it still does not provide a spanning tree in which more powerful nodes are placed higher up in the tree. Also, the algorithm requires significant state management at each node for constructing the spanning tree. Our approach, in contrast, is self-managing, with limited reliance on fixed well-known servers, requires a moderate amount of state information at each node, provides fast and efficient operation, and explicitly includes support for availability, security, and anonymity.

## Conclusion and future steps

In this paper, we have described SRIRAM, a system that automates the process by which applications can be replicated in a distributed environment. Any application whose availability is improved by the presence of replicas can benefit from such an automated mechanism. This would include applications that are based on relatively static (or slowly changing) data and do not require very stringent synchronization of the data that they use. The bulk of applications that are available today over the Web, including applications for personalization and transformation of content, fall into this category. The applications that do not benefit from replications are those that operate on highly volatile data and require strict syn-

chronization among the operation of replicas, or use such voluminous data that automated replication becomes too inefficient.

There are several issues that need to be addressed for improving the efficacy of the architecture. First, we are evaluating several alternatives for maintaining hints at the hint-servers. Our goal is to design better hint allocation schemes in which few hints become obsolete and which result in better spanning tree formation. Second, we are addressing the performance issues in the context of a node joining or leaving the mesh: the overhead in updating the tree and accounting for dynamic membership of nodes. Third, we are exploring a better replication scheme which is not purely push-based, but rather a hybrid scheme that combines it with pull-based replication in order to reduce the overhead.

We are currently in the process of implementing a prototype of this architecture and refining the architecture so that it can provide enhanced functions in the future. Some of the functions that we want to incorporate in an extended architecture include the means for authentication of the software running SRIRAM, schemes to bypass portions of the spanning tree in a search, and methods to support multiple spanning trees with different roots. We are also looking at other applications of the SRIRAM architecture, such as the development of highly scalable directory services.

**Trademark or registered trademark of Sun Microsystems, Inc.

## Cited references


1. F. Dabek et al., "Building Peer-to-Peer Systems with Chord," *Proceedings of the Eighth IEEE Workshop on Hot Topics in Operating Systems* (HotOS-VIII), IEEE, New York (2001).
2. The Gnutella Protocol Specification, Lime Wire LLC, http://www9.limewire.com/developer/gnutella_protocol_04.pdf.
3. Akamai Technologies, Inc., FreeFlow content distribution service, http://www.akamai.com.
4. Mirror Image® Internet, Inc., *insta*Content® Global Distribution Services, http://www.mirrorimage.net/services/contentdistribution.html.
5. I. Foster, C. Kesselman, J. Nick, and S. Tuecke, "The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration," http://www.globus.org/research/papers/ogsa.pdf.
6. P. Francis, Your Own Internet Distribution, http://www.aciri.org/yoid/.
7. C. Ellison et al., "SPKI Certificate Theory," Internet Engineering Task Force, RFC 2693, September 1999, http://www.ietf.org/rfc/rfc2693.txt.
8. T. Dierks and C. Allen, "The TLS Protocol Version 1.0," Internet Engineering Task Force, RFC 2246, January 1999, http://www.ietf.org/rfc/rfc2246.txt.
9. S. Kent and R. Atkinson, "Security Architecture for the Internet Protocol," Internet Engineering Task Force, RFC 2408, November 1998, http://www.ietf.org/rfc/rfc2408.txt.
10. G. A. Marin, C. P. Immanuel, P. F. Chimento, and I. S. Gopal, "Overview of the NBBS Architecture," *IBM Systems Journal* **34**, No. 4, 564–589 (1995).
11. E. Cerami, *Web Services Essentials*, O'Reilly & Associates, Sebastopol, CA (February 2002).
12. See http://jakarta.apache.org/tomcat/.
13. Napster protocol specification, http://opennap.sourceforge.net/napster.txt.
14. UDDI project, *The UDDI Technical White Paper*, http://www.uddi.org/.
15. D. Pendarakis, S. Shi, D. Verma, and M. Waldvogel, "Almi: An Application Level Multicast Infrastructure," *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems* (USITS), USENIX, Berkeley, CA (2001), pp. 49–60.
16. I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong, "Freenet: A distributed anonymous information storage and retrieval system," *Proceedings of the ICSI Workshop on Design Issues in Anonymity and Unobservability*, Berkeley, CA, June 2000, http://freenet.sourceforge.net.
17. S. Radhakrishnan, G. Racherla, C. N. Sekharan, N. S. V. Rao, and S. G. Batsell, "DST—A Routing Protocol for Ad Hoc Networks Using Distributed Spanning Trees," IEEE Wireless Communications and Networking Conference, September 1999, IEEE, New York (1999).
18. K. Psounis, "Active Networks: Applications, Security, Safety, and Architectures," *IEEE Communications Surveys* **2**, No. 1 (First Quarter 1999).
19. I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan, "Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications," *Proceedings of ACM SIGCOMM*, August 2001, San Diego, CA, ACM, New York (2001).
20. S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A Scalable Content-Addressable Network," *Proceedings of ACM SIGCOMM*, August 2001, San Diego, CA, ACM, New York (2001).
21. I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong, "Freenet: A Distributed Anonymous Information Storage and Retrieval System," *Designing Privacy Enhancing Technologies: International Workshop on Design Issues in Anonymity and Unobservability*, *Lecture Notes in Computer Science*, Vol. 2009, H. Federrath, Editor, Springer, New York (2001), pp. 46–66.
22. Y. Chu, S. Rao, and H. Zhang, "A Case for End-System Multicast," *Proceedings of ACM SIGMETRICS*, July 2000, Santa Clara, CA, ACM, New York (2000).
23. Y. Chawathe, S. McCanne, and E. A. Brewer, "*An Architecture for Internet Content Distribution as an Infrastructure Service*," February 2000, unpublished work.


*Accepted for publication August 29, 2002.*

**Dinesh C. Verma** *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598 (electronic mail: dverma@us.ibm.com).* Dr. Verma received a B.Tech. degree in computer science from the Indian Institute of Technology, Kanpur, India, in 1987 and a Ph.D. degree in computer science from the University of California, Berkeley, in 1992. Since then he has worked at the IBM Thomas J. Watson Research Center and Philips Research Laboratories. He is currently a research manager at the IBM Research Center and oversees research in the area of edge networking. His current research interests include content distribution networks, policy-based networking, and performance management in networked systems.

**Sambit Sahu** *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598 (electronic mail: sambits@us.ibm.com).* Dr. Sahu is a research staff member in the Networking Software and Services group in IBM Research. He received his Ph.D. degree in 2001 from the University of Massachusetts, Department of Computer Science. Since joining IBM, his research has focused on overlay-based communication, content distribution architecture, and design and analysis of high-performance network communication protocols. He has published a number of papers on differentiated services, multimedia, and peer-to-peer communication.

**Seraphin Calo** *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598 (electronic mail: scalo@us.ibm.com).* Dr. Calo is a research staff member at IBM Research. He received the M.S., M.A., and Ph.D. degrees in electrical engineering from Princeton University, Princeton, New Jersey. He has worked, published, and managed research projects in a number of technical areas, including: queuing theory, data communications networks, multiaccess protocols, expert systems, and complex systems management. He has been very active in international conferences, particularly in the systems management area. His current research interests include: content distribution networks, distributed applications, services management, and policy-based computing.

**Anees Shaikh** *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598 (electronic mail: aashaikh@watson.ibm.com).* Dr. Shaikh is a research staff member in the Networking Software and Services group in IBM Research. He received his Ph.D. degree in 1999 from the University of Michigan, Department of Computer Science and Engineering. Since joining IBM, his research has focused on Internet services infrastructure, particularly the areas of wide-area load balancing, performance measurement of Web-based applications, and content distribution architecture. He has published a number of papers on load-sensitive routing, middleware for real-time communication, and multicast routing.

**Isabella Chang** received her B.S. degree in 1986 and the M.S. degree in optics in 1986, both from the University of Rochester. From 1998 to 2002, she worked at the IBM Thomas J. Watson Research Center on topics related to computer communication networks with focus on implementation of network control software, such as IP quality-of-service Resource Reservation Protocols (RSVP), and DSML (Directory Services Markup Language) gateways.

**Arup Acharya** *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598 (electronic mail: arup@us.ibm.com).* Dr. Acharya is a research staff member in the Edge Networking group at the IBM Thomas J. Watson Research Center. His research interests include emerging network architectures such as VoIP, MPLS, and IPv6, as well as mobile wireless networking. Before joining IBM, he was with NEC C&C Research Laboratories in Princeton, New Jersey, between May 1995 and November 1999. He received a B.Tech. degree in computer science from the Indian Institute of Technology, Kharagpur, and a Ph.D. degree from Rutgers University in 1995. He has published numerous papers in his areas of interest and has been a program committee member of leading technical conferences in mobile networking.

# Autonomic service deployment in networks

by R. Haas
P. Droz
B. Stiller

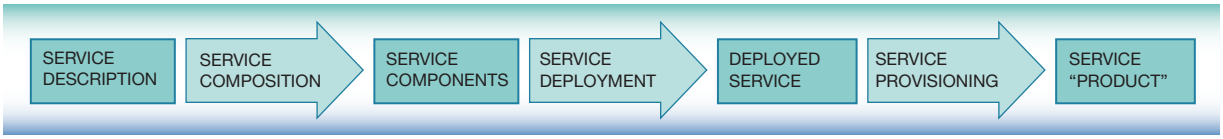
Networks have been growing dramatically in size and functionality in past years. Internet Protocol network nodes not only forward datagrams using longest-prefix matching of the destination address, but also execute functions based on dynamic policies such as proxy-caching, encryption, tunneling, and firewalling. More recently, programmable behaviors have begun to appear in network elements, allowing experimentation with even more sophisticated services. This paper presents an autonomic approach to network service deployment that scales to large heterogeneous networks. Topological categories of service deployment are introduced. A two-phase deployment mechanism that is split into hierarchically distributed and central computations is presented and illustrated with examples of actual services in a programmable network environment, together with their deployment algorithms and simulation results. Autonomic service deployment allows the distributed and complex capabilities present in network elements to be leveraged more efficiently when installing new services than is possible in traditional centralized network management-based approaches. As a result, installation is faster and use of functional resources is more optimized.

A network manager faces a daunting task today when designing, configuring, and provisioning a complete service for customers, and when trying to obtain the most use of the specific capabilities available in sophisticated network elements such as programmable routers, encryption and transcoding gateways, traffic shapers and purifiers, and distributed caches, just to name a few. However, it would not be profitable to add more capabilities to a network, for instance, in the form of network processors,[1] unless they can be exploited efficiently when installing and running a service.

If we consider an environment of networks with large numbers of nodes that have widely varying capabilities and resources and that need to be enabled with new services, it is necessary to define and provide a way to organize the deployment of new services at both the network and the node levels. The framework presented here addresses both levels globally, as well as the interactions taking place between them.

Activities that focus on the deployment of services over heterogeneous programmable networks are still very few and do not focus on those aspects that are exacerbated in large networks. Policy-based networking allows a high-level policy to be transformed into lower-level network-node configurations.[2] Such mechanisms depend on an efficient resource discovery and enablement, as presented here. Dynamic composition and deployment of services in the context of end-to-end application sessions are addressed in Refer-

Figure 1 Network service operations




ences 3, 4, and 5. This applies, for instance, to the setup of a network path for a multimedia session based on the availability and cost of image transcoders and compression service components active in intermediate network nodes. Active networks[6] achieve self-controlled deployment of services in a network by embedding service execution code into data packets so that the service remains dedicated to that flow of packets. This method is particularly suitable for environments with many network nodes that support the necessary execution environment and for short-lived flows that require an *ad hoc* deployment of a service exclusively along the path through which the flows have been routed. Particularly well-suited for large-scale problems are hierarchical architectures that have been used in the context of routing protocols and network management but not yet considered for deploying services.

To accelerate the deployment of network services, at least at the node level, efforts have begun focusing on the standardization of interfaces within networking equipment, either in the form of control protocols for label switches (Internet Engineering Task Force [IETF] General Switch Management Protocol [GSMP][7]), Internet Protocol (IP) routers (IETF ForCES[8]), and media gateways (IETF MEGACO[9]), or more generic application programming interfaces (APIs) such as those described in References 10 and 11. Therefore, it is expected that in a network a variety of solutions are likely to coexist.

Although the work presented here specifically addresses network services, the deployment of higher-level services such as Web services, for which the network can be viewed as a black box, indirectly benefits from the underlying network service-deployment framework.

The next section of this paper first presents the network-level and node-level service-deployment phases, then classifies the types of services supported by the framework presented here, and finally reviews the key elements such as the representation of capabilities and the hierarchical architecture. The third section focuses on network-level deployment. It presents a formalism for hierarchically distributed computations, illustrated with examples and algorithms. Simulation results of the network-level deployment are presented in the fourth section.

## Service-deployment framework

Service deployment denotes the set of tasks required to provide a new service dynamically in a partially or fully programmable network. A service is an assembly of components that have to be identified and placed appropriately in a network. Service provisioning is the task that operates on a service already deployed in order to provide a product of that service. For instance, encrypted flows are a product of the Virtual Private Network (VPN) service, and the VPN service is a product of its components present in the network nodes, performing encryption or decryption at the edges and quality-of-service (QoS) in the intermediate nodes, as shown in Figure 1. Whereas service composition defines the components required by a service and how to compose them, service deployment performs the actual mapping of these components into the network.

Clearly, providing tailored services means that new components have to be placed adequately in the network. We argue that an *autonomic* approach is the only scalable solution to service deployment, given the heterogeneity and size of today's networks as well as the variety of different services and the frequency at which such services have to be deployed. Autonomic means that the network itself orchestrates the deployment process, and the interaction with the network manager is limited to specifying the service according to customer needs.

More specifically, this framework splits service deployment into two successive phases, namely, *macro* and *micro deployment*. As shown in Figure 2, each phase covers a certain scope of the network, and the border between these scopes can be adjusted. In the

Figure 2 Macro and micro deployment

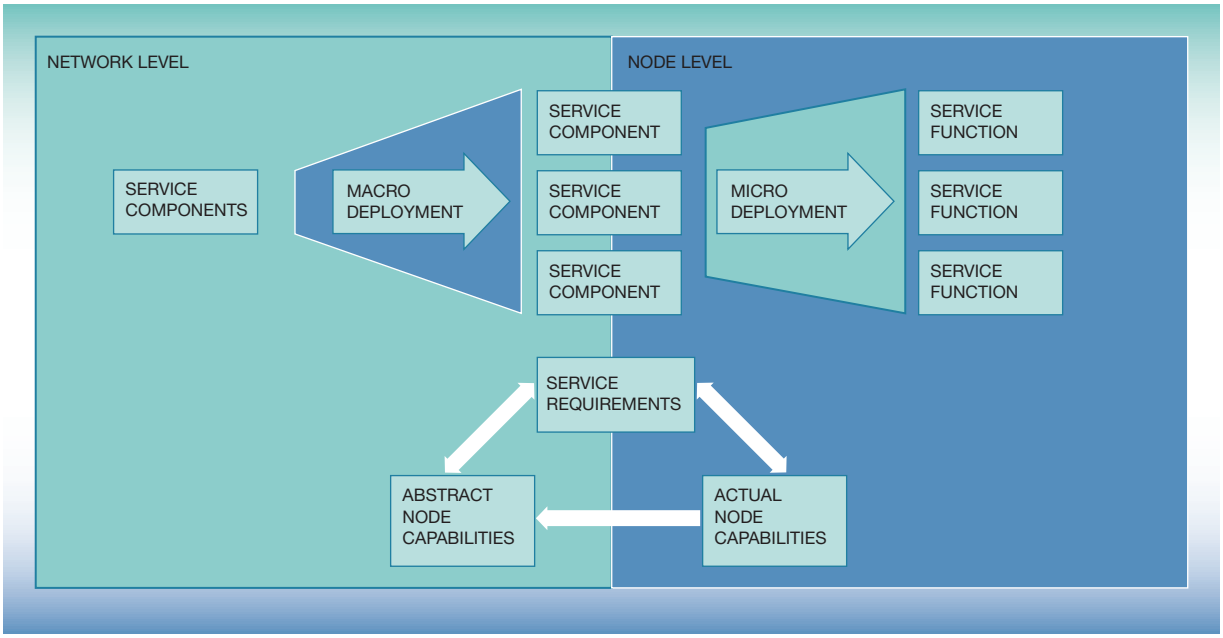



Table 1 Relevant characteristics of macro and micro deployment

| Macro Deployment | Micro Deployment |
|---|---|
| Network level: | Node level: |
| Distribute hierarchically | Distribute centrally (control point) |
| Use abstract representation of node capabilities | Use actual representation of node capabilities |
| Minimize nodes' resources for matchmaking, get Faster processing Solicit more nodes | Make use of specific node capabilities when installing the service, for Fine-grained optimization |

following discussion, we choose to place the border at the distributed-router level. Therefore, the macro deployment has a network-wide scope, whereas the micro deployment has a node-local scope. This choice does not preclude other scenarios in which the border is set instead at the local-area-network level, for instance.

For macro or network-level deployment, a sequence of five steps is executed in a hierarchically distributed manner, as described in more detail in the next section. For micro or node-level deployment, a centralized resource co-allocation method is used that benefits from information gathered during the network-level phase in order to place functions optimally. A service component could need resources of different types to be allocated, one for each service function constituting the service, hence the co-allocation problem. The main characteristics of both phases are summarized in Table 1.

**Categories of services.** Services are assumed to be decomposable into sets of components to be executed by individual nodes. We distinguish the following topological categories of service deployment and provide examples of current network services:

- *Path-based*, between a set of source(s) and destination(s), which is further divided into two types:
  - *Continuous*, for which the same component must be present in *each* node on the path, for instance, application-specific queuing (such as IETF Differentiated Services, or diffserv) that has to be enabled on all nodes of a path
  - *Sparse path-based*, or discontinuous, for which a set of components must be present in a *set of nodes* on the path. This type can be, for instance, a mul-

timedia transcoding and compression service, with one node on the path performing transcoding while another node performs compression.

- *Fence-based*, orthogonal to path-based, for which nodes along a path (possibly a loop) must act on the traffic crossing them, such as a firewall spanning multiple access routers
- *Node-based*, for which only selected nodes need to be activated, and no source or destination pairs are specified, but rather domains, such as a transparent Web cache acting for a group of end stations
- *Combinations* of the above, such as a path-and-node-based VPN service with encryption at the endpoints and QoS support in the intermediate nodes

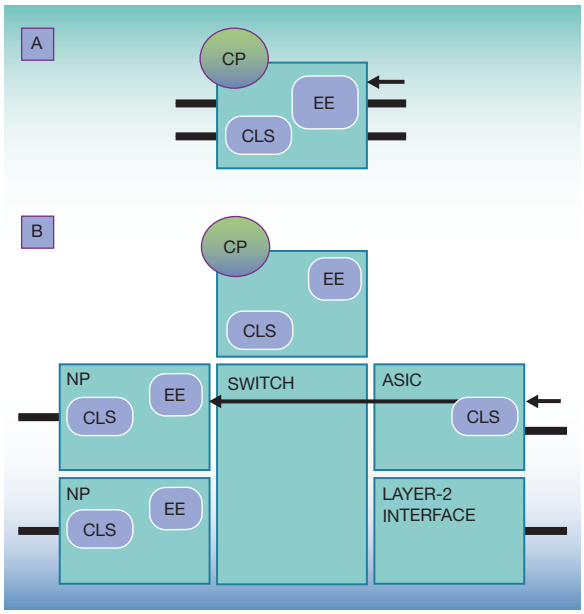
Whenever necessary, service redundancy can be achieved by deploying the service along multiple distinct paths or nodes.

**Description of service requirements and node capabilities.** Topological properties help to classify services from a network-level connectivity point of view, but a larger set of properties is required to fully describe service requirements as well as network or node capabilities. These properties are the following:

- Topology—description of the connectivity, which can be modified dynamically such as in wavelength-switching networks or with hot-pluggable node modules
- Functionality—description of functions, which can be static, configurable, or even programmable
- Performance—measure of resources, such as bandwidth or delay in networks and CPU speed in nodes
- Cost—administrative measure for using the above resources, relevant when the economical dimension must be taken into account during deployment, addressed in Reference 12.

The representation of node capabilities can, for instance, be expressed in XML (Extensible Markup Language) together with the appropriate Schema, as an extension to the IETF MIBs (Management Information Bases) such as described in References 13 and 14. It includes a description of the type of APIs to access, configure, and operate the resources in the node, either base[10] or higher-level resources such as operating-system-resident services, as well as the utilization of these resources. A similar representation for services is matched against node capabilities. This representation includes the required node capabilities to accommodate the service. The actual evaluation process of node capabilities against service requirements exceeds the scope of this paper. The information resulting from this evaluation is a metric or a set of metrics that are used to direct the deployment of the service onto the most appropriate nodes, as is illustrated later in this paper. When this information is transported toward the top of the service-deployment hierarchy, during the network-level deployment phase, it can be aggregated differently, depending on service needs. The service-deployment framework proposed lets services themselves define the information they require to execute their deployment and how it should be aggregated. This defining can be viewed as an application of active code for the network control plane.

Figure 3 (A) Abstract capabilities, (B) actual capabilities for a programmable distributed router




Network processors (NPs) are one of the key building blocks for a programmable network infrastructure. They have widely varying capabilities, such as the number of simultaneous forwarding tables supported (required for VPN support), hardware assists (required for fast packet handling), and software-level programmability (required for flexible packet handling).

Figure 3 is a simple illustration of the abstract and actual representations of capabilities, as described in Figure 2. Part A of Figure 3 shows the abstracted relevant capabilities of the distributed router described in Part B, namely, a control point (CP), an execution environment (EE), and a packet-classifier mechanism (CLS). If a tunneling service that requires encapsulation and decapsulation of data packets to be performed at line speed has to be deployed, the macro deployment phase will select an appropriate endpoint for the tunnel, based on the abstract view of Part A. Once it is known that this particular node is the most appropriate one and which interface is to be used for this service (indicated by an arrow in Part A), micro deployment is able to proceed and allocate the necessary functions, such as the EE to perform the encapsulation and decapsulation, and the CLS to select the relevant packets to be tunneled. For instance, the CLS capability in the application-specific integrated circuit (ASIC) of the interface selected and the EE of an NP can be used, depending on availability.

Figure 4 shows a short example of possible NP capabilities. Using XML for such a representation rather than an MIB-like structure is interesting because XML is easily extendable, thanks to its self-contained structure.

**Service-deployment hierarchy.** Compared to existing hierarchies for routing or network management, the service-deployment hierarchy extends the summarization (or aggregation) techniques to treat more generic information than only IP or ATM (asynchronous transfer mode) addressing and QoS. Whereas network management mostly performs collection and aggregation of data upwards, the service-deployment hierarchy is used in both directions: to collect data resulting from the service requirements versus node capabilities evaluation and to execute the deployment of a service based on the data collected. Note that although the number of hierarchy levels is not limited by the mechanism, the resources in the network to maintain this hierarchy are bounded. ATM and IP networks commonly use three levels of hierarchy to aggregate routes: two levels for intranetwork routing, such as IP OSPF (Open Shortest-Path First) areas and Autonomous Systems, and a third level for internetwork routing, such as BGP (Border-Gateway Protocol). The hierarchy may extend downwards in network nodes such as distributed routers (clusters) that to the outside appear as a single node with a single IP address. Placing the border between macro and micro deployment within such nodes can bring the advantages of macro deployment to automating the placement of functions in large clusters (see Table 1).

The physical network topology is the main factor in creating a hierarchy, at least in fixed networks. A spanning tree is built by successively grouping nodes at each hierarchy level. Figure 5 shows a simplified example of a seven-node network on top of which a three-layer hierarchy has been built. Nodes B.1, B.2, and B.3 are grouped together and represented by logical node B at the next level of the hierarchy. Routing across a hierarchical network requires the use of uplinks,[15] as represented for node B in thick dashed lines (B.1 − A), (B.2 − A), and (B.3 − C) [uplinks (A.1 − B), (A.2 − B), and (C.1 − B) are not shown]. For instance, when routing from a source in A to a destination in C, uplink (B.3 − C) shows that B.3 is the only possible border node toward C, whereas in the opposite direction, uplinks (B.1 − A) and (B.2 − A) show that B.1 and B.2 are the only possible border nodes toward A. For certain types of services, proper deployment requires that nodes along entire paths are enabled with a certain service component. In such cases, the use of uplinks is necessary.

## Network-level service deployment

This section concentrates on the network-level deployment procedure and its formalization. We then illustrate how it is implemented for service deployment of the path-based and path-and-node-based categories, together with specific algorithms used in the deployment steps.

**Deployment procedure.** From a high-level perspective, the network-level service-deployment procedure can be broken down into five steps. Figure 6 shows these steps and the resulting deployment procedures when all or only some of the steps are executed. Using only the last two steps in Figure 6 leads to a *manual deployment and automatic configuration* of a service. This is how services are generally deployed in networks today. With the intermediate solution, for example, skipping the first step, the result is an *automatic deployment with generic metrics and automatic configuration*. Only when all five steps are executed will an *automatic deployment with custom metrics and automatic configuration* result.

Computations are distributed in a logical hierarchy described in Reference 16. Layers in the hierarchy are built by recursively grouping logical or physical

Figure 4 XML representation of the capabilities of a network processor

```
<Network_Processor>
  <base_capabilities>
    <API_supported> ForCES, NPF </API_supported>
    <general>
      <processing>
        <speed> 500 MHz </speed>
      </processing>
      <scheduling>
        <total_bandwidth> 100 Mbit/s </total_bandwidth>
        <type> WFQ </type>
        <max_queues> 1000 </max_queues>
      </scheduling>
      <buffers_management>
        <total_buffer_size> 1 MB </total_buffer_size>
        <max_buffer_pools> 16 </max_buffer_pools>
        <buffer_sharing> yes </buffer_sharing>
        <Advanced Queue Management> RED </AQM>>
      </buffers_management>
      <forwarding>
        <type> layer-4 </type>
        <fields> source destination address port </fields>
        <line_rate> 100% </line_rate>
        <table_size> 100k </table_size>
        <number_of_tables> 1 </number_of_tables>
      </forwarding>
    </general>
    <resource_usage>
      // current usage for the defined capabilities
    </resource_usage>
  </base_capabilities>
  <diff_serv>      // absent if NP does not provide
                   // explicit support for diffserv
    <API_supported> ForCES, NPF </API_supported>
      <general>
        <classifier>
          <fields> 6 </fields>
          <options> ranges_support </options>
          // etc
        </classifier>
         // etc
      </general>
      <resource_usage>
        // current usage for the defined capabilities
      </resource_usage>
  </diff_serv>
  // etc
</Network_Processor>
```

nodes and representing them by a single logical node at the next layer of the hierarchy.

During the solicitation step, service requirements are distributed to nodes, following the service-deployment hierarchy downwards (see Figure 5). The summarization step collects the metrics created at the lowest level of the hierarchy, namely the physical nodes, and successively hands them over to the next higher layer of the hierarchy, possibly repeatedly aggregating the results to prevent information overflows. Once the result reaches the top level of the hierarchy, the dissemination step can start, in which the metrics collected at each level are evaluated and

Figure 5 A sample three-layer hierarchy

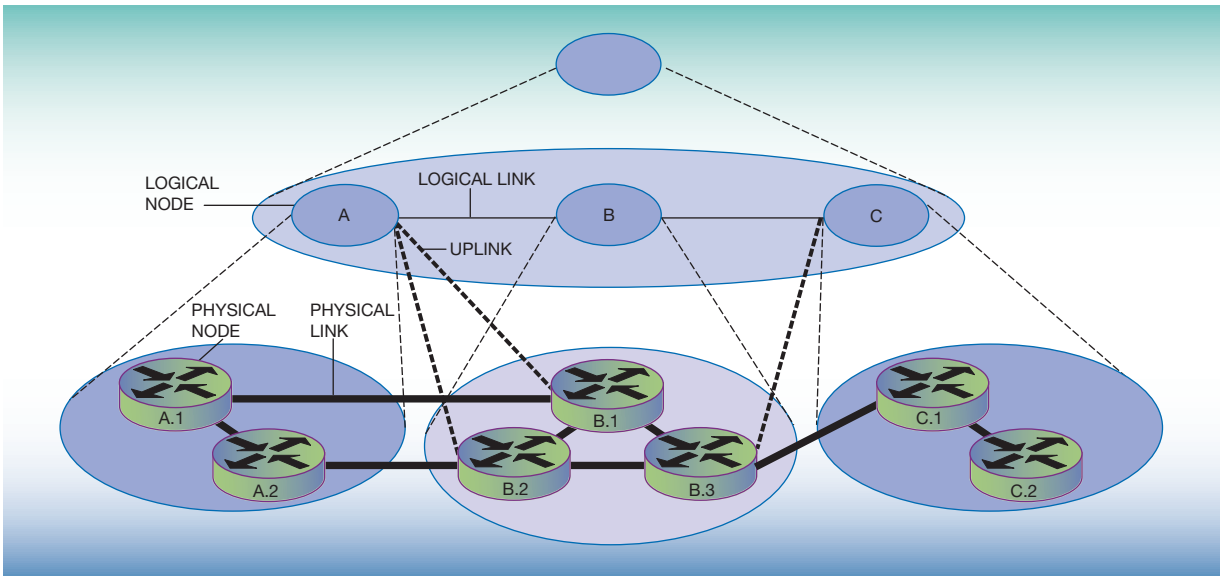
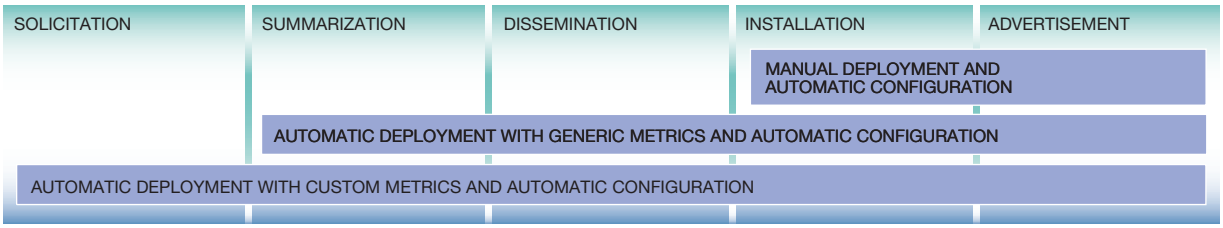



Figure 6 The five steps of the service-deployment mechanism




the most appropriate nodes are chosen until the physical nodes are reached. In these nodes the installation step loads the service. The final advertisement step mimics the summarization step, albeit only involving nodes that have been chosen to run the service.

This procedure can also be viewed as two successive query-reply processes taking place over the entire network, the queries going down the hierarchy and the replies going upwards, in which the nodes at the intermediate layers actively process the queries and replies. The first query-reply pair is used to extract the information from the network on how it can support a certain service (solicitation and summarization steps), whereas the next query-reply pair is used to actually install the service in the manner most appropriate (dissemination, installation, and advertisement steps).

**Hierarchical iterative gather-compute-scatter algorithm.** The procedure described in the previous subsection with its five steps can be divided into a sequence of iterations, each consisting of gather, compute, and scatter phases, distributed over the service-deployment hierarchy, hence the name HIGCS (hierarchical iterative, gather-compute-scatter). HIGCS uses an approach similar to the one in Reference 17, albeit with specific enhancements. A possible implementation of such a mechanism could involve mobile agents whose navigation model is extracted from the structure of the hierarchy. We

first describe the formal model and then how it is used to express the service-deployment procedure.

Service-deployment HIGCS messages are exchanged at each iteration, following the tree-like topology of the underlying hierarchy. The sets of destinations and origins relevant for the messages exchanged in the scatter and gather phases, respectively, are obtained from the compute phase. The logical node of a group is merely responsible for communicating with its underlying peer-group members of the scatter set. The logical node has neither to monitor all members of its group nor perform all computations centrally. For ease of description, we assume in the following discussion that the scatter set is always determined by a central computation performed by the logical node rather than distributing this computation among group members.

Each node executes iterative computations based on a tuple $\{(G_i, C_i, S_i)|0 \leq i \leq (k-1)\}$, where $k$ is the total number of iterations. The gather set $G_i$ is defined here as the set of (logical or physical) nodes from which messages are expected. The compute phase $C_i$ executes once the *iMsg* messages have been received from all nodes in $G_i$, as illustrated in Figure 7. The scatter set $S_i$ denotes the (logical or physical) nodes to which *oMsg* messages are sent once the compute phase $C_i$ completes. $oMsg_n$ messages can differ, depending on their destination node $n$ in the $S_i$ set. Similarly to that in Reference 17, node attributes are assumed to be available during the compute phase. This includes, for instance, the hierarchy level at which the node is located.

The generic signaling-message format used by the network-level service-deployment mechanism is defined in Table 2.

To perform service deployment, iterations are associated with the steps as described in Figure 6. For that purpose, we define the following general behaviors for $C_i$:

- $C_0$ selects the set of underlying nodes $S_0$ that have to be solicited (null set if executed on a physical node).
- $C_1$ summarizes information gathered from the set of underlying nodes $G_1$ (on a physical node, this information is created) and places it in *sMetric*.
- $C_2$ selects the set of nodes $S_2$ where the service is to be deployed (on a physical node, the service is installed).
- $C_3$ summarizes the results from the deployment on the set of nodes $G_3$ (on a physical node, this information is obtained locally) and places it in *iMetric*.

Figure 7 The HIGCS agent operation model

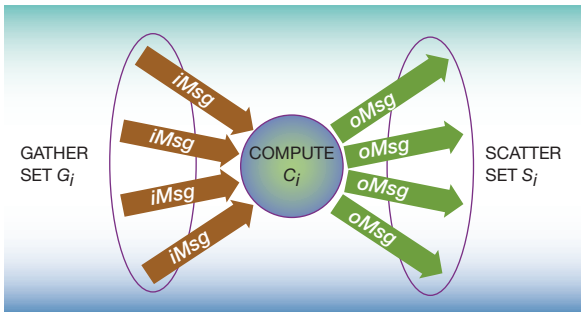



Table 2 The HIGCS service-deployment message format

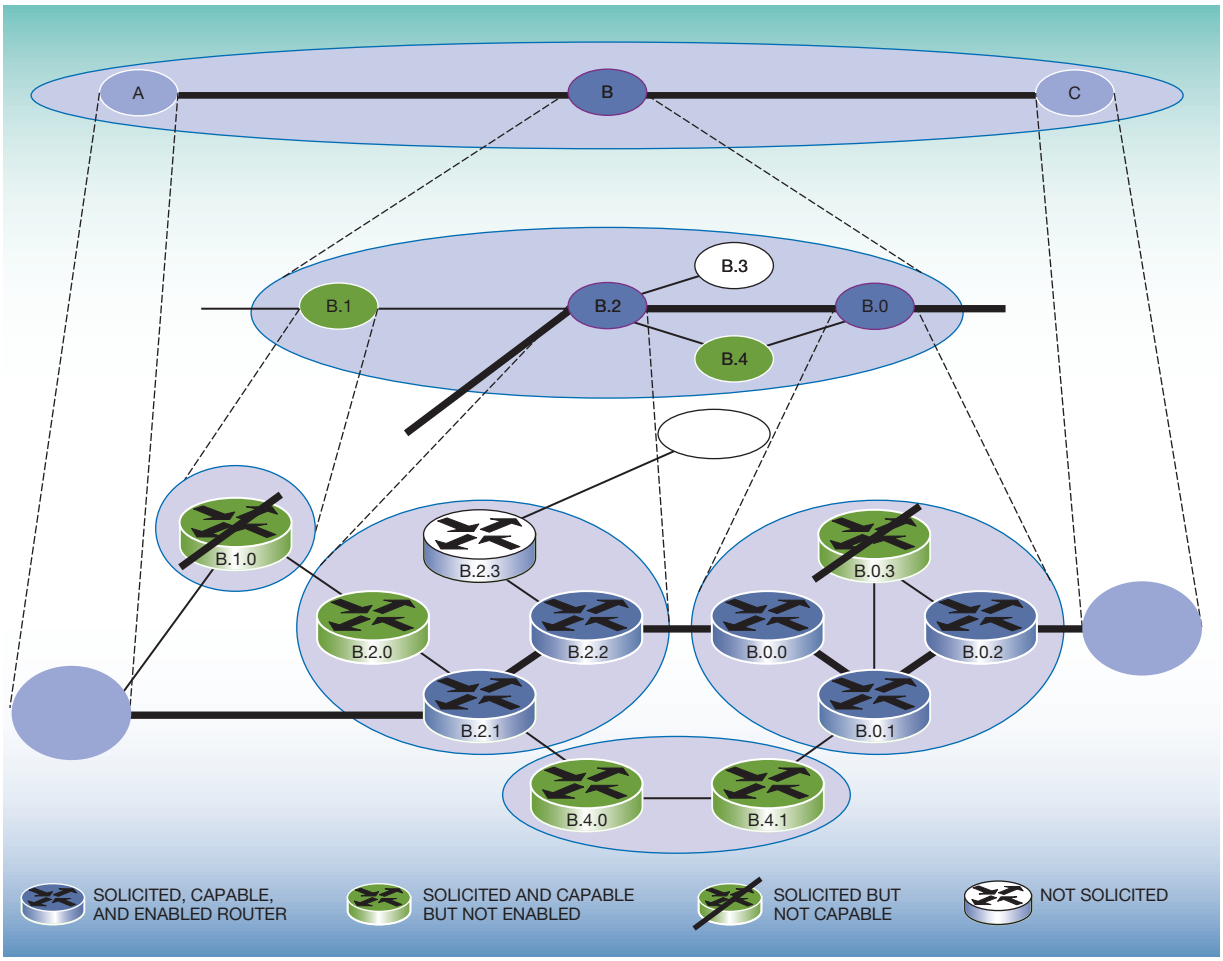
| Parameter | Generic Value |
|---|---|
| *serviceId* | Instance of service deployed |
| *hierarchyId* | Identifier of service hierarchy |
| *srcId* | Source of message |
| *destId* | Destination |
| *iterationId* | Current iteration |
| $G_i$ | Gather set for iteration $i$ |
| $C_i$ | Compute function for iteration $i$ |
| $S_i$ | Scatter set for iteration $i$ |
| *servSpec* | Service specification |
| *sMetric* | Solicited metric |
| *iMetric* | Installed metric |
| . . . | Other service-specific information |

The actions executed by nodes in the hierarchy during the first query-reply pair, namely, the solicitation and summarization steps, are expressed in the $C_0$ and $C_1$ compute phases, whereas $C_2$ and $C_3$ correspond to the second query-reply pair, namely, the dissemination, installation, and advertisement steps.

Clearly, these $C_i$ functions will be implemented differently for each service to be deployed, as we describe next. More detailed examples of the HIGCS iterations and $C_i$ functions for each category of service deployment are given in Reference 18.

**Path-based deployment.** Table 3 shows the computations executed for the deployment of a hypothetical path-based service on all nodes of a path between two customer sites (represented by the A and C top-level nodes in Figure 8). For the sake of simplicity, the straightforward update of other fields in the

Figure 8 Result of the HIGCS computations




*oMsg* (such as *srcId*) and fields that do not change from the *iMsg* message are not shown.

Figure 8 illustrates the result after all four HIGCS iterations have completed across the entire hierarchy. The path selected is shown as a thicker line. The first iteration was initiated by node B, which received the initial HIGCS message with *iMsg.ends* set to {A, C}. Logical node B.3 and router B.2.3 did not participate in the service deployment, since they are not on a path between A and C. All other nodes had been solicited, and nodes B.0.3 and B.1.0 were excluded, since their capabilities did not match the requirements of that service. Of the remaining routers, B.0.0, B.0.1, B.0.2, B.2.1, and B.2.2 are finally enabled with the service, since they are on the shortest path between A and C.

For path-based deployment, routing decisions have to be taken successively at each layer of the hierarchy (executed by SelectNodesOnShortestPath in Table 3). Among the various suitable data representations, transition matrices offer an accurate view of the cost of traversing a logical node.

Transition matrices used in path-based service deployment contain topology-, capability-, performance-, or cost-related information, or all of these types of information. For instance, the sMetric generated for node B in Figure 8 is the following transition matrix:

$$\mathcal{T}_B = \begin{pmatrix} 1 & \cdots & \\ 0 & 0 & \cdots \\ 5 & 0 & 1 \end{pmatrix}$$

Elements $t_{i,j}$ indicate that there is connectivity between border nodes B.$i$ and B.$j$, with a path composed of nodes capable of running that particular service. In addition, the value of $t_{i,j}$ corresponds to the performance of that path in terms of number of hops. For instance, element $t_{0,2}$ in $\mathcal{T}_B$ with a value of 5 indicates that there is a path composed of capable nodes when crossing B from B.0 to B.2 (and vice versa). This path is shown with a thicker line at each hierarchy level in Figure 8, which consists of five nodes (or hops) at the bottom hierarchy layer.

***Straight-path search.*** In path-based deployment, it is only necessary to solicit those nodes that form a path between the endpoints specified. Nodes that lie in stub networks, as, for instance, B.3 in the example of Figure 8, do not have to be solicited. In addition, assuming that node capabilities are available in the same way irrespective of the input and output port on a node (this normally applies to physical nodes), it is possible to further constrain the set of possible paths as follows: Assume $S$ is the set of all possible paths without loops between two endpoints. $S_s$ contains all paths from $S$ for which no subpath exists in $S$. (A path $P_s$ is defined as a subpath of another path $P$ if by removing one or more nodes from the list of nodes that form $P$ we obtain the list of nodes that form $P_s$.) We define such paths in $S_s$ as *straight paths*. Clearly, the shortest path is a straight path. Note also that straight paths are not transitive. Appending a straight path between nodes b and c, such as {b, c}, to a straight path between a and b, such as {a, e, b}, does not necessarily lead to a straight path between a and c, as {a, e, c} is a subpath of {a, e, b, c}, as shown in Figure 9. In this example, the only straight paths between a and c are {a, d, b, c} and {a, e, c}.

Here, we present a novel algorithm that searches for straight paths (SPS) and performs similarly to the Depth-First Search (complexity of $O(n^2)$, where $n$ is the total number of nodes). The algorithm discovers all nodes that lie on a straight path between two endpoints *source* and *dest* in a graph $G$ composed of nodes $V[G]$, where $Adj[u]$ is the set of nodes adjacent to node $u$.

```
SPS(source,dest)
 1  for each u ∈ V[G] do
 2     color[u] ← WHITE
 3     marked[u] ← FALSE
 4  SPS-Visit(source,dest)
```

Figure 9 Example network for SPS

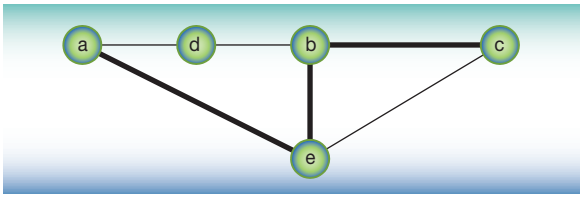



Table 3 $C_0$, $C_1$, $C_2$, and $C_3$ functions

| | | |
|---|---|---|
| $C_0$ | | DS-solicit |
| $S_0$ | ← | SelectAllNodesBetween (*iMsg.ends*) |
| $oMsg_n.ends$ | ← | SelectNeighborNodes ($n \mid n \in S_0$) |
| $G_1$ | ← | $S_0$ |
| $C_1$ | | DS-summarize |
| $S_1$ | ← | GetLogicalNode ( ) |
| $oMsg.sMetric$ | ← | **if** IsLogicalNode ( ) **then** SummarizeMetrics ($iMsg_j.sMetric, \forall j \in G_1$) **else** CreateMetric (*iMsg.servSpec*) |
| $G_2$ | ← | $S_1$ |
| $C_2$ | | DS-disseminate |
| $S_2$ | ← | **if** IsLogicalNode ( ) **then** SelectNodesOnShortestPath (*iMsg.ends*) **else** null |
| $oMsg_n.ends$ | ← | SelectNeighbornodes ($n \mid n \in S_2$) |
| $G_3$ | ← | $S_2$ |
| $C_3$ | | DS-install |
| $oMsg.iMetric$ | ← | **if** IsLogicalNode ( ) **then** SummarizeInstalledMetrics ($iMsg_j.iMetric, \forall j \in G_3$) **else** InstallService (*iMsg.servSpec*) |
| $S_3$ | ← | GetLogicalNode ( ) |

```
SPS-Visit(u,dest)
 1  localresult ← 0
 2  color[u] ← GRAY
 3  for each v ∈ Adj[u] do
 4     if v = dest then
 5        localresult ← 2
 6  if localresult < 2 then
 7     for each v ∈ Adj[u] do
 8        if color[v] = WHITE then
 9           if NumGrayNeighbors(v) < 2 then
10              localresult ← max(SPS-Visit(v,dest),
                   localresult)
11           else localresult ← max(localresult,1)
12  if localresult = 2 then
13     marked[u] ← TRUE
```

```
14  color[u] ← WHITE
15  return(localresult)

NumGrayNeighbors(u)
 1  result ← 0
 2  for each v ∈ Adj[u] do
 3     if color[v] = GRAY then
 4        result ← result + 1
 5  return (result)
```

In the initialization phase of the algorithm (lines 1–3 of SPS( )), all nodes in the graph are marked as FALSE and colored WHITE. Nodes that lie on a straight path between *source* and *dest* will be marked by the SPS algorithm. The color of each node can alternate between WHITE and GRAY as nodes are being visited. Note that nodes may be visited multiple times on different paths.

The principle of the algorithm is to grow all possible paths in the graph recursively and temporarily color nodes on such paths gray, until one of the two following conditions applies: the path either reaches the destination, or the path reaches a node that is adjacent to a gray node (therefore already in the same path). This is the case if NumGrayNeighbors($v$) is equal to 2, on line 9 of SPS-*Visit*( ). If the first condition applies, then all nodes on this path are marked as belonging to a straight path (line 13 of SPS-*Visit*( )). In both cases, the color reverts from gray to white as the algorithm continues and recursively searches for other paths. It stops once all possible paths have been visited.

As a brief proof that all nodes on straight paths and only those nodes are indeed marked by this algorithm, we use the following property: If any subpath of some path is not a straight path, then the path itself is not a straight path either. Note that a node can belong to many of the paths in $S$ simultaneously, including paths in the subset $S_s$, but only those nodes that belong to at least one path of $S_s$ will be marked. By construction, the algorithm extends all paths from *source* to *dest*, except those paths it abandons at some intermediate node because the subpath from *source* to that intermediate node is not straight (and hence, according to the property above, any extension of that subpath to *dest* would not lead to a straight path). Therefore, all paths that have been successfully extended to *dest* comprise the set $S_s$ of all possible straight paths between *source* and *dest*. By construction, the algorithm marks only those nodes—but all of them—that belong to each such path in $S_s$, thereby marking certain nodes possibly more than once. As a result, all nodes on straight paths are marked at least once by the algorithm.

A performance improvement[19] to the algorithm consists in reusing the marking obtained at each node. Because straight paths are not transitive, this might lead to false markings; therefore, more nodes than necessary might be solicited.

**Path- and node-based deployment.** As an example of a service deployed both along paths and at selected nodes with differing requirements, we consider the deployment of a VPN. Encryption capabilities are required at the VPN endpoints, and QoS treatment of packets is ensured by RSVP-enabled (IETF Resource Reservation Protocol, or RSVP) nodes between those endpoints.

To accommodate both path- and node-based characteristics, we choose to extend the transition matrix presented in the previous subsection with the necessary node information. The extended transition matrix is defined as follows:

$$\mathcal{T}_{\mathrm{N}} = (\mathcal{M}; \mathcal{P})$$

Elements $m_{i,j}$ indicate the shortest number of RSVP-capable hops between border nodes N.$i$ and N.$j$ in node N. Elements $p_{i,j}$ indicate the shortest path from a node in domain $D_j$ that fulfills the requirements of the VPN-endpoint service specification to border node $i$. The VPN interconnects $n$ domains $D_n$, which are represented by logical nodes.

Figure 10 shows a group of nodes with their capabilities. For simplicity, it is assumed that all VPN-endpoint-capable nodes are also RSVP-capable, but not vice versa. The extended transition matrix for this group is:

$$\mathcal{T}_{\mathrm{A.1}} = \left(\begin{array}{cccc|c} 1 & \ldots & & & 1 \\ 0 & 0 & \ldots & & 0 \\ 4 & 0 & 1 & \ldots & 3 \\ 2 & 0 & 3 & 1 & 1 \end{array}\right)$$

In $\mathcal{T}_{\mathrm{A.1}}$, element $m_{2,0}$ indicates that the number of hops to cross A.1 from A.1.2 to A.1.0 is four. Element $p_{2,0}$ indicates that the number of hops to reach a VPN-capable endpoint in A.1 entering from A.1.2 with a path composed of RSVP-capable nodes only is three. There are actually two such paths, namely, (A.1.2, A.1.4, A.1.3) and (A.1.2, A.1.6, A.1.3).

If we assume that logical node A.1 represents one domain and that logical node A.2 (not shown) represents another domain, then the $\mathcal{P}$ matrix of the extended transition matrix for logical node A, composed of A.1 and A.2, will have two columns $p_{:1}$ and $p_{:2}$, one for each domain. When summarizing such extended matrices, a new column is appended for each domain considered.

***Hierarchical Steiner-tree construction.*** When constructing a VPN, it might be relevant to minimize the total cost of interconnecting the VPN endpoints. The Minimum Steiner Tree is the minimum-cost tree that interconnects such endpoints, and is an NP-complete problem in the case of additive edge weights. Several algorithms approximate the optimum solution, such as Selective Closest Terminal First (SCTF[20]), which finds a solution guaranteed to be at most twice as expensive as the optimum tree. Here, we address how Steiner trees can be built over hierarchical networks.

During the summarization step, as the locations of the branching points of the Steiner tree are not yet known, only the shortest paths between all pairs of nodes at that hierarchy level are computed and stored as transition matrices in the sMetric. It can be shown that extending each transition matrix with the costs of all Steiner trees that can be constructed between all combinations of three or more border nodes would incur a large amount of additional transported information and computation effort in the summarization step. In addition, the SCTF algorithm cannot operate with the resulting interdependent edge costs.

Based on this information, a Steiner tree is computed successively at each hierarchy level during the subsequent dissemination step, for instance, using the SCTF algorithm. The cost used in a computation at a given level is an upper bound of the actual cost of the Steiner tree that can be built at the level below. This can lead to suboptimum decisions being made, as shown in Figure 11. Here, nodes A.1 and A.2 have identical transition matrices, and therefore the algorithm will select one of these two nodes indistinctively to become the branching point for the Steiner tree in node A. The cost of this tree, indicated in Figure 11 by a thicker line in node A, is forecast to be seven, which is the upper bound given the cost of four for traversing node A.2 according to its transition matrix. While the algorithm repeats in node A.2, it computes a local Steiner tree of cost three, leading to a final cost of six for the overall Steiner tree, instead of the upper bound of seven that would have been reached if node A.1 had been selected instead of A.2.

Figure 10 A group of nodes solicited for the VPN service

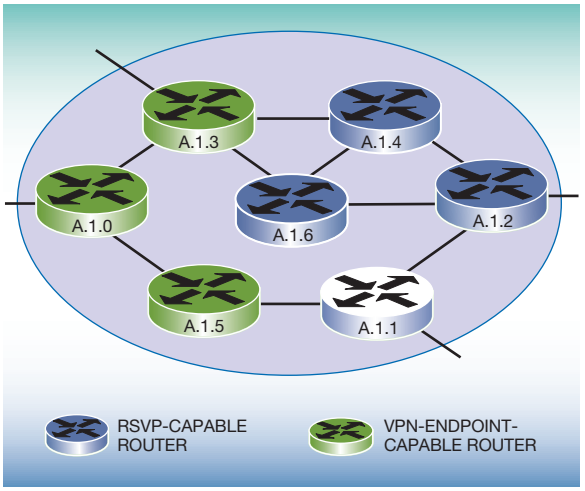
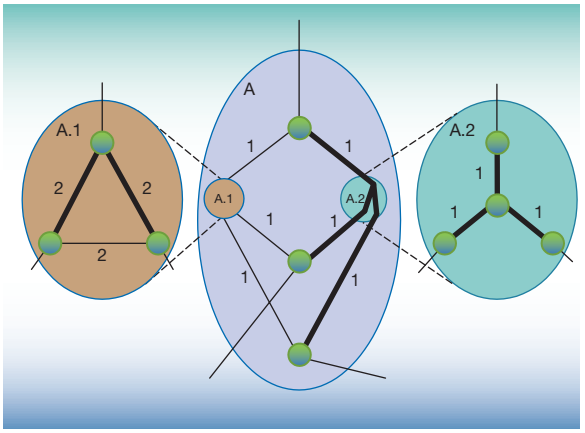
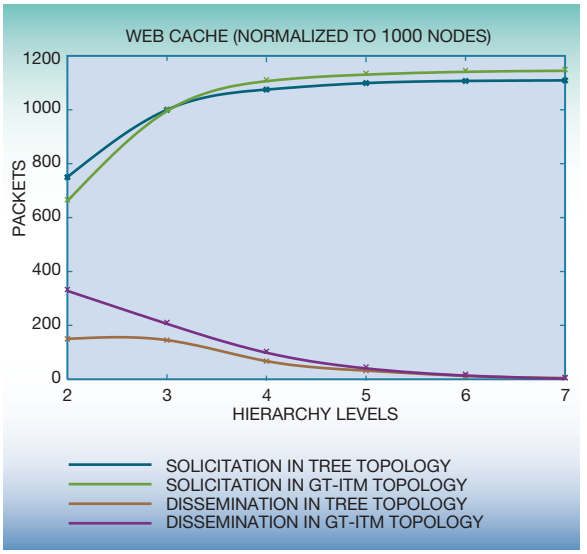



Figure 11 Hierarchical construction of a Steiner tree




The information-aggregation method used by transition matrices is designed so that the shortest path can be accurately computed over a hierarchical network as presented earlier, but it lacks information that would allow the construction of better Steiner trees as described above. This method illustrates the necessary trade-off between acceptable quality and amount of information that has to be found for each

Figure 12 Total HIGCS messages for Web-cache deployment




service to be deployed, hence the complexity of the processing it entails.

## Simulation results

The elements and algorithms of network-level service deployment presented in the previous section have been implemented in a simulation environment to evaluate their scalability over large networks. The HIGCS agent-based deployment protocol has been implemented as an extension to the discrete-event network simulator ns-2. We present here some simulation results obtained with various fixed network topologies and for two types of services:

- A Web-cache node-based service, for which the Web cache is deployed in the most appropriate physical node in each lowest-layer peer group (B.0, B.1, B.2, B.3, and B.4, in Figure 8)
- A three-endpoint diffserv path-based service, using the directed broadcast mechanism shown earlier for the solicitation step, the Floyd-Warshall all-pairs shortest-path algorithm for the summarization step, and the Steiner-tree computation shown previously for the dissemination step.

Whereas typical Internet topologies consist of three routing-hierarchy levels, we have simulated the service-deployment protocol over topologies ranging from a flat two-level up to a highly hierarchical seven-level topology, as produced by the GT-ITM (Georgia Tech Internetwork Topology Models) topology modeler. The average size of peer groups is fixed to three nodes at every level so as to obtain comparable results for the various topologies, and also to keep the maximum number of nodes ($\sum_{n=0}^{6} 3^n = 1093$) within the simulator capabilities. For each discrete number of hierarchy levels, ten different topologies are generated using a random placement of nodes on a grid, and connectivity is determined with the Waxman probabilistic method.[21] Random numbers are generated using the Stanford GraphBase pseudo-random-number generator.[22]

As a benchmarking topology, a tree-network topology that ideally fits the hierarchical structure of HIGCS is also simulated. In this topology, HIGCS messages that travel between logical nodes only go over a single physical hop.

Interesting measures of the overhead incurred by the service-deployment protocol are the total number of messages exchanged during the five deployment steps and the total compute time spent in all nodes participating in the procedure.

Figure 12 shows the total number of HIGCS messages sent over the topology (normalized to a 1000-node topology) for the deployment of the Web-cache service. The standard deviation is zero for the tree topology (number of packets is deterministic) and very small for the randomly generated GT-ITM topologies. Note that despite the arbitrary logical-node selection policy applied in the simulations, the GT-ITM and the tree topologies show close to identical results. The more hierarchy levels, the more solicitation packets are transmitted, since every logical or physical node receives such a packet, but this increase becomes less and less important. During the dissemination step, the overhead is negligible because only one node in each lowest-level peer group has to be reached.

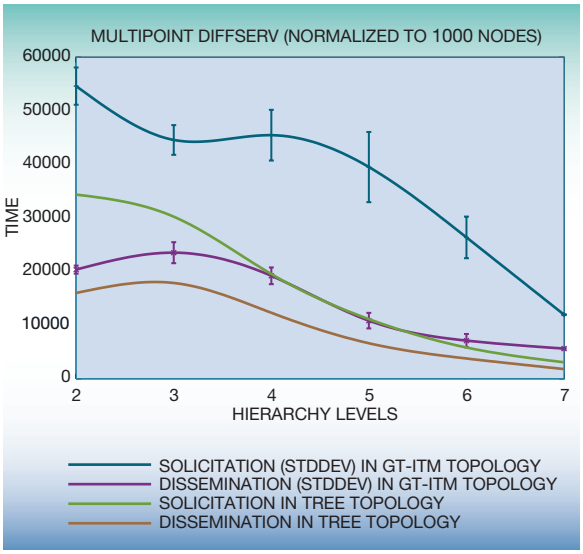
Figure 13 shows the total compute time (defined as a relative measure on normalized topologies) for the deployment of a three-endpoint diffserv service. Standard deviation is shown only for the ten randomly generated GT-ITM topologies for each number of hierarchy levels, since the measurement variations in the fixed-tree topologies are negligible.

The *divide and conquer* method takes effect as it can be seen that compute time decreases noticeably when

more hierarchical levels are introduced, because in turn the set of nodes that each logical node is responsible for decreases.

These results confirm the influence of the number of levels in the deployment hierarchy. The total compute time, shown here for the solicitation and dissemination steps only, decreases when more hierarchy levels are introduced in a given topology, whereas only a minor increase in message overhead is incurred. Hierarchical network-level deployment based on the suitable algorithms for the service-deployment categories presented therefore allows an efficient and scalable allocation of services in a network.

Figure 13 Total HIGCS compute time for diffserv deployment




## Conclusion

Organizing the autonomic deployment of services is a key step toward an intelligent network infrastructure. Openness and programmability are starting to appear in network equipment, and it is therefore necessary to provide generic mechanisms that can enable the deployment of any type of service.

In this paper, we have introduced a two-phase mechanism that achieves an efficient and flexible deployment of services in networks: a macro-level deployment phase that operates in a hierarchically distributed manner to query and collect capabilities of nodes in the network, and then execute the deployment itself; and a micro-level deployment phase that refines the actual installation of a service according to the specific capabilities of each element comprising the network node.

In addition, we have introduced various categories of services according to topological service-deployment needs and proposed novel information aggregation methods, as well as a straight-paths graph algorithm used during the macro-level deployment phase that allows the number of nodes queried to be limited. The trade-off between accuracy, amount of information, and availability of efficient algorithms required for near-optimum deployment was illustrated using the hierarchical Steiner-tree computation.

By making the network itself responsible for executing the deployment of a service, an autonomic deployment is achieved, avoiding time-consuming and error-prone manual operations. As networks grow in size and heterogeneity, the deployment mechanism can still capture the essential data for deploying any particular service, while retaining its scalability thanks to the use of summarization and dissemination across the service-deployment hierarchy, as confirmed by our simulations.

## Cited references


1. *Power Network Processor Software Overview*, Technical Report, IBM Corporation (April 2002); see http://www.chips.ibm.com/techlib.
2. D. Verma, M. Beigi, and R. Jennings, "Policy-Based SLA Management in Enterprise Networks," *Proceedings of the Workshop on Policies for Distributed Systems and Networks (POLICY 2001)*, Bristol, UK (2001), *Lecture Notes on Computer Science*, Vol. 1995, Springer, Berlin (2001), pp. 137–152.
3. K. Nahrstedt, D. Wichadakul, and D. Xu, "Distributed QoS Compilation and Runtime Instantiation," *Proceedings of IEEE/IFIP 8th International Workshop on Quality of Service (IWQoS'2000)*, Pittsburgh, PA (June 5–7, 2000).
4. A. Nakao, L. Peterson, and A. Bavier, "Constructing End-to-End Paths for Playing Media Objects," *Proceedings of OpenArch 2001—The 4th IEEE Conference on Open Architectures and Network Programming*, Anchorage, AK (March 2001).
5. S. Choi, J. Turner, and T. Wolf, "Configuring Sessions in Programmable Networks," *Proceedings of INFOCOM 2001*, Anchorage, AK (April 2001), pp. 60–66.
6. D. Wetherall, U. Legedza, and J. Gutta, "Introducing New Internet Services: Why and How," *IEEE Network: The Magazine of Global Information Exchange* **12**, No. 3, 12–19 (1998).
7. A. Doria, F. Hellstrand, K. Sundell, and T. Worster, *General Switch Management Protocol V3*, IETF RFC 3292 (June 2002); see http://www.ietf.org.
8. ForCES, *Requirements for Separation of IP Control and For-*

*warding*, IETF draft <draft-ietf-forces-requirements-05.txt> (June 2002).
9. N. Greene, M. Ramalho, and B. Rosen, *Media Gateway Control Protocol Architecture and Requirements*, IETF RFC 2805 (April 2000); see http://www.ietf.org.
10. J. Biswas et al., *Application Programming Interfaces for Networks*, Technical Report, IEEE PIN1520 Working Group (2000); see http://www.ieee-pin.org.
11. Software Working Group, Network Processing Forum, Fremont, CA (2002), at www.npforum.org.
12. R. Haas, P. Droz, and B. Stiller, "Cost- and Quality-of-Service-Aware Network-Service Deployment," *Proceedings of Advanced Internet Charging and QoS Technology (ICQT 2001)*, Vienna, Austria (September 2001), pp. 166–171.
13. P. Grillo and S. Waldbusser, *Host Resources MIB*, IETF RFC 1514 (September 1993); see http://www.ietf.org.
14. F. Baker, K. Chan, and A. Smith, *Management Information Base for the Differentiated Services Architecture*, Internet Engineering Task Force (IETF) Request for Comments (RFC) 3289 (May 2002); see http://www.ietf.org.
15. *Private Network-Network Interface Specification Version 1.0 (PNNI V1.0)*, af-pnni-0055.000, the ATM Forum (March 1996); see http://www.atmforum.com/pages/aboutatmtech/approved.html.
16. R. Haas, P. Droz, and B. Stiller, "A Hierarchical Mechanism for the Scalable Deployment of Services over Large Programmable and Heterogeneous Networks," *Proceedings of the IEEE International Conference on Communications (ICC 2001)*, Helsinki, Finland (June 11–15, 2001), pp. 2074–2078.
17. Y. Chae, S. Merugu, E. Zegura, and S. Bhattacharjee, "Exposing the Network: Support for Topology-Sensitive Applications," *Proceedings of OpenArch 2000—The 3rd IEEE Conference on Open Architectures and Network Programming*, Tel Aviv, Israel (March 26–27, 2000).
18. R. Haas, P. Droz, and B. Stiller, "Distributed Service Deployment over Programmable Networks," *Proceedings of the 12th International Workshop on Distributed Systems: Operations & Management (DSOM'01)*, Nancy, France (October 2001), pp. 113–128.
19. A. Kumar and R. Haas, *Design and Implementation of a Distributed-Agent-Based Simulation for Hierarchical Service-Deployment*, Technical Report RZ 3378, IBM Corporation, Zurich Research Laboratory, Rüschlikon, Switzerland (2001).
20. S. Ramanathan, "Multicast Tree Generation in Networks with Asymmetric Links," *IEEE/ACM Transactions on Networking* **4**, No. 4, 558–568 (1996).
21. E. W. Zegura, K. L. Calvert, and S. Bhattacharjee, "How to Model an Internetwork," *Proceedings of INFOCOM 1996*, Vol. 2 (March 1996), pp. 594–602.
22. D. E. Knuth, *The Stanford GraphBase: A Platform for Combinatorial Computing*, Addison-Wesley Publishing Co., Reading, MA (1994).

*Accepted for publication August 9, 2002.*

**Robert Haas** *IBM Research Division, Zurich Research Laboratory, Säumerstrasse 4, CH-8803 Rüschlikon, Switzerland (electronic mail: rha@zurich.ibm.com).* Mr. Haas received the M.S. degree in communication systems from the Swiss Federal Institute of Technology (EPFL), Lausanne, and the Eurecom Institute, Sophia-Antipolis, France, in 1996. He also received the D.E.A. degree in distributed systems from the University of Nice Sophia-Antipolis. He joined the IBM Thomas J. Watson Research Center in 1996 as a research staff member, designing and prototyping a layer-3 switch. In 1998 he joined the Zurich Research Laboratory, and he is currently studying for a Ph.D. degree with the Swiss Federal Institute of Technology (ETHZ), Zurich. His research interests include network protocols and architecture, specifically auto-configurable and programmable networks.

**Patrick Droz** *IBM Research Division, Zurich Research Laboratory, Säumerstrasse 4, CH-8803 Rüschlikon, Switzerland (electronic mail: dro@zurich.ibm.com).* Dr. Droz received an M.S. degree in computer science from the Swiss Federal Institute of Technology (ETHZ), Zurich, in 1992. He then joined the Zurich Research Laboratory and worked in the ATM Networking Group on the design and implementation of the ATM control point for the 8260 campus backbone hub, and completed his Ph.D. degree on "Traffic Estimation and Resource Allocation in ATM Networks" in 1996. He is now manager of the Network Processor Software group. He is cochair of the IETF ForCES working group. His current research activities focus on software enablement for network processors.

**Burkhard Stiller** *Information Systems Laboratory IIS, University of Federal Armed Forces Munich, D-85579 Neubiberg, Germany, and also Computer Engineering and Networks Laboratory TIK, ETH Zurich, CH-8092 Zurich, Switzerland (electronic mail: stiller@informatik.unibw-muenchen.de).* Prof. Dr. Stiller received an M.S. degree in computer science and a Ph.D. degree from the University of Karlsruhe, Germany, in 1990 and 1994, respectively. He worked there as a research assistant at the Institute of Telematics until 1995, and was a visiting scientist at the University of California, Irvine, California, in 1992, and at the University of Cambridge, Computer Laboratory, in the United Kingdom in 1994/1995 under a European Community Research Fellowship. From 1995 until 1999 he was with the Computer Engineering and Networks Laboratory (TIK), Swiss Federal Institute of Technology (ETHZ), Zurich, as a research associate and lecturer for multimedia communications. Since August 1999 he has been an assistant professor for communication systems at ETHZ. In addition, he was appointed full professor for computer science at the University of Federal Armed Forces, Munich, Germany, in April 2002.

# Enabling autonomic behavior in systems software with hot swapping

by J. Appavoo, K. Hui, C. A. N. Soules, R. W. Wisniewski, D. M. Da Silva, O. Krieger, M. A. Auslander, D. J. Edelsohn, B. Gamsa, G. R. Ganger, P. McKenney, M. Ostrowski, B. Rosenburg, M. Stumm, J. Xenidis

Autonomic computing systems are designed to be self-diagnosing and self-healing, such that they detect performance and correctness problems, identify their causes, and react accordingly. These abilities can improve performance, availability, and security, while simultaneously reducing the effort and skills required of system administrators. One way that systems can support these abilities is by allowing monitoring code, diagnostic code, and function implementations to be dynamically inserted and removed in live systems. This "hot swapping" avoids the requisite prescience and additional complexity inherent in creating systems that have all possible configurations built in ahead of time. For already-complex pieces of code such as operating systems, hot swapping provides a simpler, higher-performance, and more maintainable method of achieving autonomic behavior. In this paper, we discuss hot swapping as a technique for enabling autonomic computing in systems software. First, we discuss its advantages and describe the required system structure. Next, we describe K42, a research operating system that explicitly supports interposition and replacement of active operating system code. Last, we describe the infrastructure of K42 for hot swapping and several instances of its use demonstrating autonomic behavior.

As computer systems become more complex, they become more difficult to administer properly. Special training is needed to configure and maintain modern systems, and this complexity continues to increase. Autonomic computing systems address this problem by managing themselves.[1] Ideal autonomic systems just work, configuring and tuning themselves as needed.

Central to autonomic computing is the ability of a system to identify problems and to reconfigure itself in order to address them. In this paper, we investigate hot swapping as a technology that can be used to address systems software's autonomic requirements. Hot swapping is accomplished either by *interpositioning* of code, or by *replacement* of code. Interpositioning involves inserting a new component between two existing ones. This allows us, for example, to enable more detailed monitoring when problems occur, while minimizing run-time costs when the system is performing acceptably. Replacement allows an active component to be switched with a different implementation of that component while the system is running, and while applications continue to use resources managed by that component. As conditions change, upgraded components, better suited to the new environment, dynamically replace the ones currently active.




0018-8670/03/$5.00 © 2003 IBM

Hot swapping makes downloading of code more powerful. New algorithms and monitoring code can be added to a running system and employed without disruption. Thus, system developers do not need to be prescient about the state that needs to be monitored or the alternative algorithms that need to be available. More importantly, new implementations that fix bugs or security holes can be introduced in a running system.

The rest of the paper is organized as follows. The next section describes how hot swapping can facilitate the autonomic features of systems software. An important goal of autonomic systems software is achieving good performance. The section "Autonomically improving performance" illustrates how hot swapping can autonomically improve performance using examples from our K42[2] research operating system (OS) as well as from the broader literature. The section that follows describes a generic infrastructure for hot swapping and contrasts it with the adaptive code alternative. Then the section "Hot swapping in K42" describes the overall K42 structure, presents the implementation of hot swapping in K42, and includes a brief status and a performance evaluation. The next section discusses related work, and the concluding section contains some final comments.

## Autonomic features through hot swapping

Autonomic computing encompasses a wide array of technologies and crosses many disciplines. In our work, we focus on systems software. In this section we discuss a set of crucial characteristics of autonomic systems software and describe how hot swapping via interposition and replacement of components can support these autonomic features, as follows.

*Performance*—The optimal resource-management mechanism and policy depends on the workload. Workloads can vary as an application moves through phases or as applications enter and exit the system. As an example, to obtain good performance in multiprocessor systems, components servicing parallel applications require fundamentally different data structures than those for achieving good performance for sequential applications. However, when a component is created, for example, when a file is opened, it is generally not known how it will be used. With replacement, a component designed for sequential applications can be used initially, and then it can be autonomically switched to one supporting greater concurrency if contention is detected across multiple processors.

*System monitoring*—Monitoring is required for autonomic systems to be able to detect security threats, performance problems, and so on. However, there is a trade-off between placing extensive monitoring in the system and the performance overhead this entails. With support for interposition, upon detection of a problem by broad-based monitoring, it becomes possible to dynamically insert additional monitoring, tracing, or debugging without incurring overhead when the more extensive code is not needed. In an object-oriented system, where each resource is managed by a different instance of an object, it is possible to garner an additional advantage by monitoring the code managing a specific resource.

*Flexibility and maintainability*—Autonomic systems must evolve as their environment and workloads change, but must remain easy to administer and maintain. The danger is that additions and enhancements to the system increase complexity, potentially resulting in increased failures and decreased performance. To perform hot swapping, a system needs to be modularized so that individual components may be identified. Although this places a burden on system design, satisfying this constraint yields a more maintainable system. Given a modular structure, hot swapping often allows each policy and option to be implemented as a separate, independent component, with components swapped as needed. This separation of concerns simplifies the overall structure of the system. The modular structure also provides data structures local to the component. It becomes conceivable to rejuvenate software by swapping in a new component (same implementation) to replace the decrepit one. This rejuvenation can be done by discarding the data structures of the old object, then starting from scratch or a known state in the new object.

*System availability*—Numerous mission-critical systems require five-nines-level (99.999 percent) availability, making software upgrades challenging. Support for hot swapping allows software to be upgraded (i.e., for bug fixes, security patches, new features, performance improvements, etc.) without having to take the system down. Telephony systems, financial transaction systems, and air traffic control systems are a few examples of software systems that are used in mission-critical settings and that would benefit from hot-swappable component support.

*Extensibility*—As they evolve, autonomic systems must take on tasks not anticipated in their original design. These tasks can be performed by hot-swapped code, using both interposition and dynamic replacement. Interposition can be used to provide existing components with wrappers that extend or modify their interfaces. Thus, these wrappers allow interfaces to be extended without requiring that existing components be rewritten. If more significant changes are required, dynamic replacement can be used to substitute an entirely new object into an existing running system.

*Testing*—Even in existing relatively inflexible systems, testing is a significant cost that constrains development. Autonomic systems are more complicated, exacerbating this problem. Hot swapping can ease the burden of testing the system. Individual components can be tested by interposing an object to generate input values and examine results, thereby improving code coverage. Delays can be injected into the system at internal interfaces, allowing the system to explore potential race conditions. This concept is motivated by a VLSI (very large scale integration) technique whereby insertion of test probes across the chip allows intermediate values to be examined.[3,4]

## Autonomically improving performance

As outlined in the previous section, autonomic computing covers a wide range of goals, one of which is improving performance. For systems software, the ability to self-tune to maintain or improve performance is one of the most important goals. In this section, we discuss how hot swapping can support and extend existing performance enhancements, allowing the OS to tailor itself to a changing environment.

**Optimizing for the common case.** For many OS resources the common access pattern is simple and can be implemented efficiently. However, the implementation becomes expensive when it has to support all the complex and less common cases. Dynamic replacement allows efficient implementations of common paths to be used when safe, and less-efficient, less-common implementations to be switched in when necessary.

As an example, consider file sharing. Although most applications have exclusive access to their files, on occasion files are shared among a set of applications. In K42, when a file is accessed exclusively by one application, an object in the application's address space handles the file control structures, allowing it to take advantage of mapped file I/O, thereby achieving performance benefits of 40 percent or more.[5] When the file becomes shared, a new object dynamically replaces the old object. This new object communicates with the file system to maintain the control information. Other examples where similar optimizations are possible are (a) a pipe with a single producer and consumer (in which case the implementation of the pipe can use shared memory between the producer and consumer) and (b) network connections that have a single client on the system (in which case data can be shared with zero copy between the network service and the client).

**Optimizing for a wide range of file attribute values.** Several specialized file system structures have been proposed to optimize file layout and caching for files with different attributes.[6,7] We can optimize the performance across the range of file attribute values by implementing a number of components, where each component is optimized for a given set of file attribute values, and then having the OS hot swap between these components as appropriate.

For example, although the vast majority of files accessed are small (<4 KB), OSs must also support large files as well as files that grow in size. Using dynamic replacement we can take advantage of the file size in order to optimize application performance. In K42, in the case of a small unshared file, the memory contents backing the file are copied to the application's address space. An object in the application's address space services requests to that file, thus reducing the number of interactions between the client and file system. Once a file grows to a larger size, the implementation is dynamically switched to another object that communicates with the file system to satisfy requests. In this case the file is mapped in memory, and misses are handled by the memory management infrastructure.

**Access patterns.** There is a plethora of literature focused on optimizing caching and prefetching of file blocks and memory pages from disk based on application access patterns.[8,9] Researchers have shown up to 30 percent fewer cache misses by using the appropriate policy. Hot swapping can exploit these policies by interposing monitoring code to track access patterns and then switching between policies based on the current access pattern.

**Exploiting architecture features.** Many features of modern processors are underutilized in today's mul-

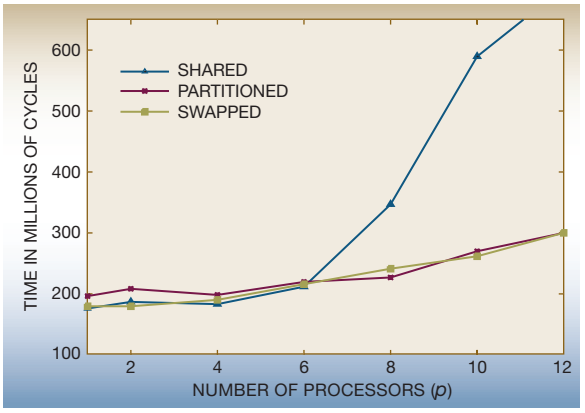
tiplatform OSs. To ensure portable code, without making global code paths unduly complex, these features are generally either crippled or ignored entirely. This is because implementers need to provide a single implementation to be used across all platforms. For example, there is only limited OS support today for large pages, even though a large number of processors support them. Hot swapping makes it easier to take advantage of such architectural features, because special-purpose objects can be introduced and used without requiring that complete functionality for all possible cases be implemented in every object.

**Multiprocessor optimizations.** In large multiprocessor systems, parallel applications can benefit from processor locality. To exploit this locality, some OSs implement services in a partitioned fashion (code is divided across different processors in order to distribute the load and avoid contention). However, these partitioned implementations consume more memory and incur larger overheads on some operations, for example, file destruction and process destruction. Conversely, shared implementations (code on one processor) can minimize space and time overheads for sequential applications.

Figure 1, which illustrates a file-searching application, can be used to visualize the performance advantages of dynamically switching between a shared and partitioned version of the objects that cache file pages in K42. The system monitors the number of application threads and switches between implementations when appropriate. The *y* axis shows the number of machine cycles to complete a sample search; lower is better. The figure shows that the shared implementation has a 10 percent performance advantage over the partitioned implementation when only one application is searching through the file on one processor. On the other hand, the shared implementation is 300 percent worse with 12 applications on 12 processors. With hot swapping, the system can dynamically switch between the two implementations and capture the best performance characteristics of each.

**Enabling client-specific customization.** Extensible OSs offer new interfaces that enable clients to customize OS components. By using components optimized for a particular application, it is possible to achieve significant performance improvements in a variety of system services.[10–13] For example, the Exokernel Cheetah Web server demonstrated factor of two-to-four increases in throughput from network

Figure 1 Processor cycles required for *p* independent concurrent searches over an entire 52 MB file (low is good)




stack and file cache specializations.[14] Hot swapping enables extensibility by allowing applications to replace OS components. Hot swapping improves upon most existing extensible systems by allowing on-the-fly switching, as well as replacement of generic system components.

**Exporting system structure information.** Technologies such as compiler-directed I/O prefetching[15] and storage latency estimation descriptors[16] have shown over 100 percent performance increases for applications, but require detailed knowledge about the state of system structures. Always gathering the necessary profiling information increases overhead and can negatively affect the performance of applications that do not require this information. Hot swapping allows selected applications to gather more information about the state of system structures by interposing a monitoring component when appropriate. By inserting these monitors only when applications will benefit, overall system performance will not degrade. Without hot swapping, the additional cost of monitoring and increased system complexity hampers the ability of researchers to consider algorithms designed for rare conditions that may be important for certain applications.

**Supporting specialized workloads.** In monolithic systems, optimizations in support of one type of workload often negatively impact the performance of other types of workload. To alleviate this problem some development groups have implemented multiple versions of an OS, where each version is

tuned for a particular type of workload. Another approach is incremental specialization,[17] where specific portions of the kernel are recompiled to optimize them for a particular type of workload. An OS using hot swapping can dynamically switch components optimized to handle these types of workload. The reported performance improvements when using incremental specialization—as high as 70 percent for small amounts of data read from files—can also be obtained using hot swapping.

## An infrastructure for hot swapping

Achieving an effective generic hot-swapping mechanism for systems software requires careful design. In addition to the impact on the surrounding system infrastructure that has to be taken into account, there are several actions involved in performing a hot swap, including triggering the hot swap, choosing the target, swapping components, transferring state, and potentially adding object types. In this section, we first describe system requirements for supporting hot swapping, which involve both interposition and replacement, and then we describe the steps involved in performing a component switch. We conclude this section by comparing hot swapping to adaptive code.

**System structure.** Many large systems, such as databases, are structured with well-defined components and interfaces to those components. This modular structure is critical for hot swapping. Well-defined interfaces are necessary for interposition and replacement of components. Any code, whether it is the kernel, a database, a Web server, or any other server or application at user level, can use the infrastructure for hot swapping. The code intended to perform the hot swap need only be structured so that there are identifiable components that can be interposed or replaced.

In a system with only global components, hot swapping can be used to change overall system performance, but it becomes difficult to tune the system to specific applications because the same component is used across all applications. Additional advantages can be gained if an object-oriented design is used, where each individual use of a resource is managed by an independent object that can be hot swapped in order to tune that resource to its workload. For example, optimization on a per-file basis is possible if each file is implemented using a different object instance that can be tuned to its access pattern.

Large parts of our existing OSs are not designed in a fashion that allows for hot swapping. However, the UNIX** Vnode[18] interface, streams facility, and device driver interface are good examples where hot swapping would be possible.

Modularity and the use of object-oriented design in OSs is expanding. Some current OS interface designs have demonstrated the effectiveness of modularity by enabling flexibility and innovation. For example, there are many Linux** file systems that have explored various possible designs behind the well-defined Linux VFS (virtual file system)[19] interface. As systems become more complex, and autonomic computing becomes more important, the incentives to adopt such designs will grow.

The rest of this paper is presented with an object-oriented structure in mind, and we use the terms "component" and "object" interchangeably. However, much of this discussion applies to systems that are not object-oriented but intend to support hot swapping.

**Performing hot swapping.** Perhaps surprisingly, only a small number of research groups have looked into hot swapping,[17,20,21] and even then, their approaches have been limited by restrictive conditions. One of the reasons may be the difficulty in providing a general and efficient service that can safely and efficiently handle interposing and replacing components on a multiprocessor in a multithreaded and preemptive environment. For demonstration purposes, consider the difficulties in replacing the TCP/IP (Transmission Control Protocol/Internet Protocol) stack. To do this requires: (1) synchronizing the switching with a potentially large number of concurrent calls from applications and various parts of the OS; (2) atomically transferring state that may include active connections, buffers, and outstanding timers; (3) installing the new object in the system so that its clients automatically and transparently use the new object instance.

The complexity of hot swapping components suggests that the implementer of a specific object will consider providing hot swapping only if the system infrastructure minimizes the implementation work needed in the individual component. Below we discuss a framework that accomplishes this, and in later sections we describe how we have implemented the infrastructure in K42.

***Triggering hot swapping.*** In many cases we expect an object itself to trigger a replacement. For example, if an object is designed to support small files and it registers an increase in file size, then the object can trigger a hot swap with an object that supports large files. In other cases, we expect the system infrastructure to determine the need for an object replacement through a hot swap. Monitoring is required for this purpose, and additional monitoring can be enabled by object interposition if more accurate information is needed before initiating the swap. For example, an OS might have a base level of monitoring in order to identify excessive paging. When this condition occurs the OS might interpose additional monitoring on objects that cache files in order to determine the source of the problem before choosing a specific object instance to replace.

In some cases, applications will explicitly request an object swap. Subsystems such as databases, Web servers, or performance-sensitive scientific applications, can choose to optimize their performance by explicitly switching in new system objects to support known demands. For example, a database application may request the system use objects that support large pages for the purpose of backing a specific region of memory.

In the future, we expect that developers of autonomic computing systems will provide the service infrastructure that allows their products to query for the latest changes, such as bug fixes and security upgrades (similar to the up2date program in Red Hat Linux 7.3). These systems will periodically download new components and hot swap them in without disrupting running applications.

***Choosing the target.*** In some cases, the initiator of a hot swap can identify the target directly as, for example, when upgrading a component. In most cases, however, the target component is more appropriately identified by its behavior than by its specific name or type. For example, a client might request a page-caching object optimized for streaming without needing to know the particular class that implements that functionality. Although introducing such a facility is relatively simple, the complexity comes both in identifying the characteristics that it should encode and the presentation of the encoding to the requester.

***Performing the swap.*** In our experience, the most complex part of hot swapping is performing the swap, including getting the object in a state suitable for swapping and performing the swap in a scalable manner (across a large number of processors). The synchronization needed to get into such a state is complex and not recommended to be implemented on a case-by-case basis. Moreover, synchronization internal to an object imposes a run-time overhead even when hot swapping is not required. In the next section we discuss the implementation of a generic hot-swap mechanism in K42.

***Transferring state.*** A key issue is how to transfer state efficiently between the source and the target of an object replacement. In many cases, the transfer of state between objects being switched is simple. For example, in K42 when an application-level object that caches file state is swapped, we invalidate cached state and pass only the control information. In other cases, the work is more involved. For example, file caching objects convert their internal list of pages into a list of generic page descriptors.

The infrastructure cannot determine what state must be transferred. It can, however, provide a means to negotiate a *best common format* that both objects support. Rather than flattening everything to a canonical state, in some cases pointers to complex data structures may be passed. This is best worked out for a given class hierarchy of objects. Additionally, on a multiprocessor, the infrastructure allows the transfer to occur in parallel.

***Dynamically adding object types.*** Downloading new code into the OS provides two challenges that need to be handled by the infrastructure. First, if an object class is changed, it is necessary to track all instances of that class in order to be able to replace those. Second, if library or subsystem code is changed, it is necessary to download code into all running applications and subsystems using that library.

**Adaptive code alternative.** Among other features, hot swapping allows system software to react to a changing environment. A more common approach in systems software to handling varying environments is to use adaptive code. Although adaptive code may not achieve the full autonomic vision previously outlined, a comparison to hot swapping is pertinent. In the simplest case, adaptive code has run-time parameters that can be modified on line. In other cases, there are multiple implementations with different data structures, and the best choice of the implementation to use can vary over time.[8,22,23] Adaptive code is a combination of several individual algorithms, each designed for a particular workload.

Table 1 Realization of autonomic features with adaptive code vs hot swapping

| Feature | Adaptive Code | Hot-Swappable Code |
|---|---|---|
| Set of possible configurations | Preprogrammed | Can be extended |
| What gets monitored | Preprogrammed | Can be extended |
| When monitoring code is in system | Always | Dynamic |
| Adaptation decision algorithm | Preprogrammed | Can be swapped |
| Code complexity | Made worse | Reduced |
| Infrastructure required | None | Significant, but once |
| Enables on-line patches | No | Yes |

Table 1 compares adaptive code to hot swapping. When used for optimizing performance, the solution involving adaptive code has three major disadvantages: required foreknowledge, higher code complexity, and higher overhead.

Adaptive code allows the system to switch between a set of preprogrammed algorithms. The set of available algorithms, the monitoring code used to gather data, and the decision-making code cannot be changed once the system is running.

Adaptive code designed for different situations, or to support many applications, is complex. This is especially true for system code designed to run across a variety of hardware platforms. Coordinating adaptation across the many components is more complicated than allowing each component to make its own adaptation decisions.

Adaptive code lacks interposition capability, and thus imposes some monitoring overhead on all requests. Achieving the right level of monitoring through both stable periods and highly loaded, unstable periods, is a challenge.

A more fundamental limitation of the adaptive code solution is its unsuitability for the larger vision of autonomic computing. For example, without the ability to add new code to the system, it does not provide a mechanism to deal with security upgrades or bug fixes. It is possible to update the code off line and restart the system, but this incurs downtime and is disruptive to applications and users.

On a case-by-case basis, the infrastructure for hot swapping we have described is more expensive than simply using specialized adaptive code. However, it only needs to be implemented once, at system development time. In contrast, adaptive systems typically have to reimplement that complexity in each of the services providing the adaptation.

Figure 2 illustrates two implementations of the same function. The adaptive code approach is monolithic and includes monitoring code that collects the data needed by the adaptive algorithm to choose a particular code path. Algorithm options must be part of the original code, and the code's overall size and complexity are increased. With hot swapping, each algorithm is implemented independently (resulting in reduced complexity per component), and is hot swapped in when needed. Monitoring code can also be interposed as needed. Decision code is decoupled from the implementation of the components. The shared code, for tracking usage patterns (not used in the random case), needs to be integrated into the code paths in the adaptive case and is inherited into each object in the hot-swapping case.

## Hot swapping in K42

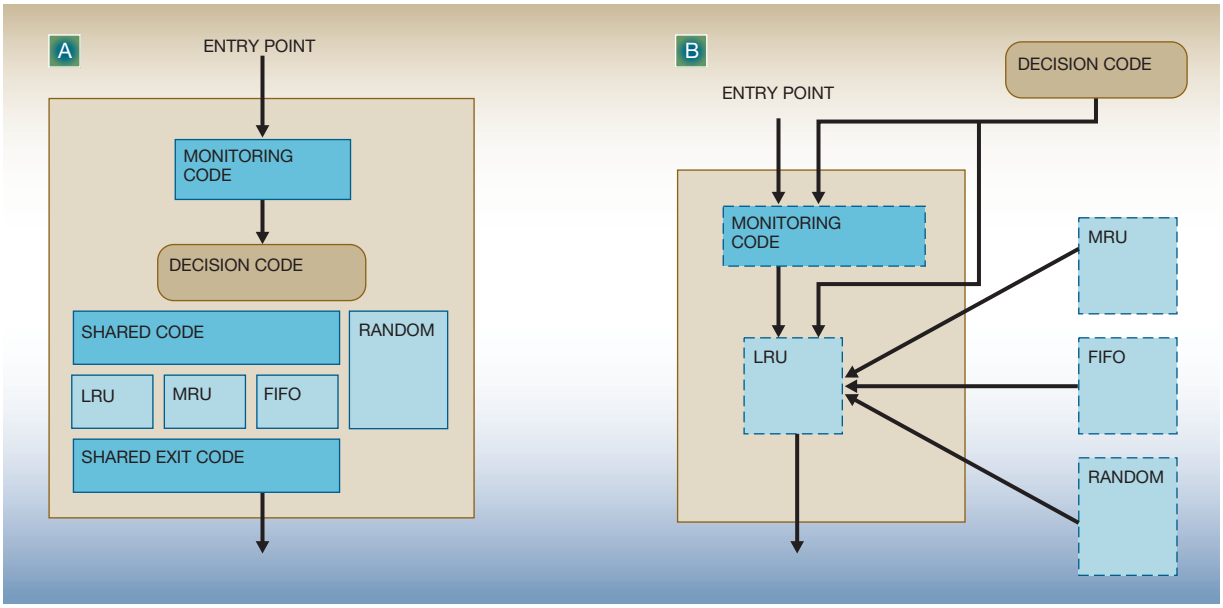
In this section, we describe the generic hot-swapping mechanism of K42. To provide context, we start by presenting the overall structure and design of K42.[2] We then describe K42's infrastructure for hot swapping, give its current status, and present some performance results.

**K42.** K42 is an open source research kernel for cache-coherent 64-bit multiprocessor systems, which currently runs on PowerPC* and MIPS** platforms, and will soon be available for x86-64 platforms.

***Project motivation and goals.*** K42 focuses on achieving good performance and scalability, on providing a customizable and maintainable system, on supporting a wide variety of platforms, systems, and problem domains, and on being accessible to a large community as Open Source Software.

- *Performance*—K42 is designed to scale up to run well on large multiprocessors and support large-scale applications efficiently. It also scales down to run well on small multiprocessors. Moreover,

Figure 2 An adaptive code implementation (A) vs a hot-swapping implementation (B) of the same function




it supports small-scale applications as efficiently on large multiprocessors as on small multiprocessors.

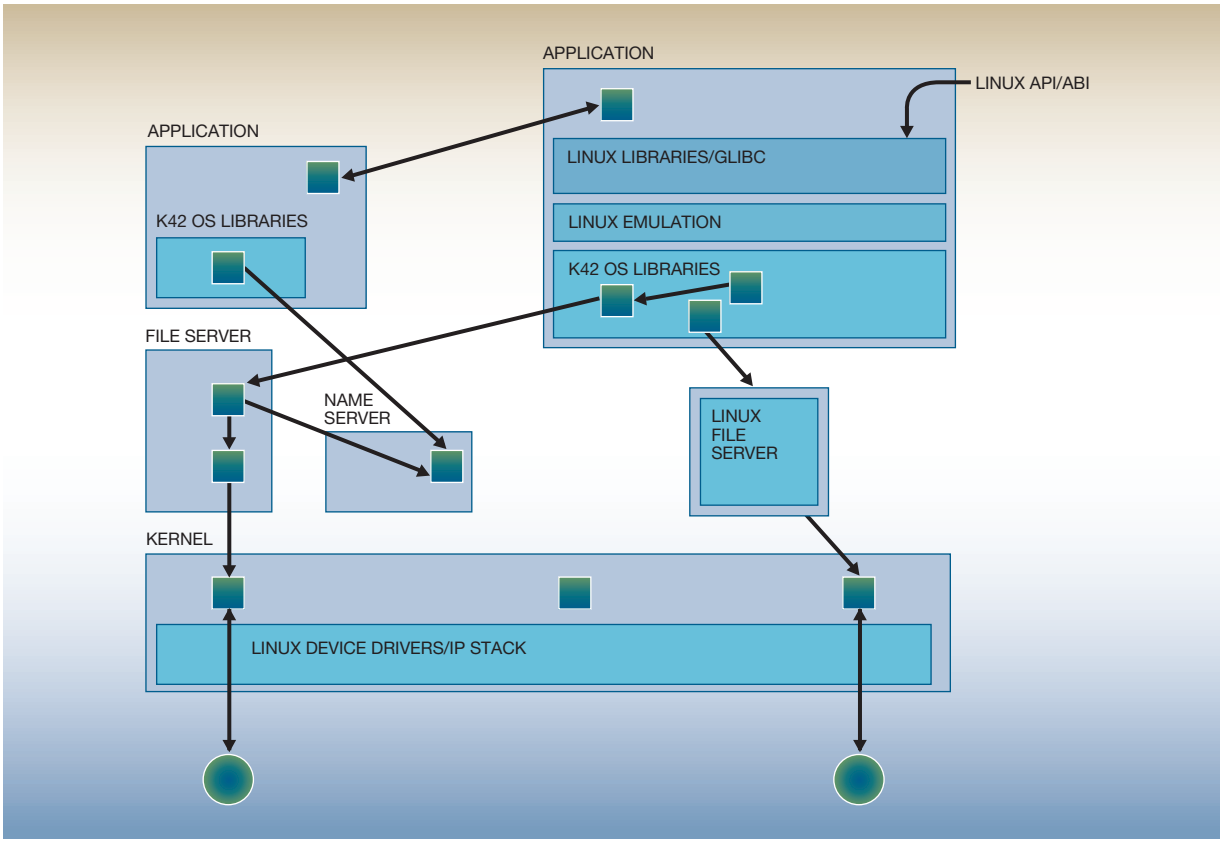
- *Customizability*—K42 allows applications to determine (by choosing from existing components or by writing new ones) how the OS manages their resources. It autonomically adapts to changing workload characteristics.
- *Applicability*—K42 is intended to effectively support a wide variety of systems and problem domains and to make it easy to modify the OS to support new processor and system architectures. It can support systems ranging from embedded processors to high-end enterprise servers.
- *Wide availability*—K42 is open source, and is available to a large community. It makes it easy to add specialized components for experimenting with policies and implementation strategies, and will open up for experimentation parts of the system that were traditionally accessible only to experts.

K42 fully supports the Linux API (application programming interface) and ABI (application binary interface) and uses Linux libraries, device drivers, file systems, and other code without modification. The system is fully functional for 64-bit applications, and can run codes ranging from scientific applications to complex benchmarks like SPEC SDET (Standard Performance Evaluation Corporation Software Development Environment Throughput) to significant subsystems like Apache Software Foundation's Apache HTTP Server. Supporting the Linux API and ABI makes K42 available to a wide base of application programmers, and our modular structure makes the system accessible to the community of developers who wish to experiment with kernel innovations.

Some research OS projects have taken particular philosophies and have followed them rigorously to extremes in order to fully examine their implications. Although we follow a set of design principles, we are willing to make compromises for the sake of performance. The principles that guide our design include (1) structuring the system using modular, object-oriented code, (2) designing the system to scale to very large shared-memory multiprocessors, (3) leveraging performance advantages of 64-bit processors, (4) avoiding centralized code paths, global data structures, and global locks, (5) moving system functionality to application libraries, and (6) moving system functionality from the kernel to server processes.

***K42 structure.*** K42 is structured around a client-server model (see Figure 3). The kernel is one of the core servers, currently providing memory management, process management, IPC (interprocess communication) infrastructure, base scheduling, net-

Figure 3 Structural overview of K42




working, device support, and so on. (In the future we plan to move networking and device support into user-mode servers.)

Above the kernel are applications and system servers, including the NFS (Network File System) file server, name server, socket server, pty (pseudoteletype) server, and pipe server. For flexibility, and to avoid IPC overhead, we implement as much functionality as possible in application-level libraries. For example, all thread scheduling is done by a user-level scheduler linked into each process.

All layers of K42, the kernel, system servers, and user-level libraries, make extensive use of object-oriented technology. All interprocess communication is between objects in the client and server address spaces. We use a *stub compiler* with decorations on the C++ class declarations to automatically generate IPC calls from a client to a server, and have optimized these IPC paths to have good performance. The kernel provides the basic IPC transport and attaches sufficient information for the server to provide authentication on those calls.

From an application's perspective, K42 supports the Linux API and ABI. This is accomplished by an emulation layer that implements Linux system calls by method invocations on K42 objects. When writing an application to run on K42, it is possible to program to the Linux API or directly to the native K42 interfaces. All applications, including servers, are free to reach past the Linux interfaces and call the K42 interfaces directly. Programming against the native interfaces allows the application to take advantage of K42 optimizations. The translation of standard Linux system calls is done by intercepting glibc (GNU C library) system calls and implementing them with K42 code. Although Linux is the first and currently

only personality we support, the base facilities of K42 are designed to be personality-independent.

We also support a Linux-kernel *internal personality*. K42 has a set of libraries that will allow Linux-kernel components such as device drivers, file systems, and network protocols to run inside the kernel or in user mode. These libraries provide the run-time environment that Linux-kernel components expect. This infrastructure allows K42 to use the large code base of hardware drivers available for Linux.

***K42 key technologies.*** To achieve the above mentioned goals, we have incorporated many technologies into K42. We have written several white papers (available on our Web site[2]) describing these technologies in greater detail; the intent of this section is to provide an overview of the key technologies used in K42. Many of these have an impact on hot swapping; for example moving functionality to application libraries makes the hot swapping infrastructure more complicated, but provides additional possibilities for customization and thus for hot swapping. K42 key technologies are listed below. The technologies used by the hot-swapping infrastructure are discussed in "K42 features used" of the subsection "Infrastructure for hot swapping."

- Object-oriented technology, which has been used in the design of the entire system, helps achieve good performance through customization, helps achieve good multiprocessing performance by increasing locality, helps increase maintainability by isolating modifications, and helps perform autonomic functions by allowing components to be hot swapped.
- Much traditional kernel functionality is implemented in libraries in the application's own address space, providing a large degree of customizability and reducing overhead by avoiding crossing address space boundaries to invoke system services.
- K42 is easily ported to new hardware and due to its structure can exploit machine-specific features such as the PowerPC inverted page table and the MIPS software-controlled TLB (translation lookaside buffer).
- Much system functionality has been implemented in user-level servers with good performance maintained via efficient IPCs similar to L4.[24]
- K42 uses *processor-specific* memory (the same virtual address on different processors maps to different physical addresses) to achieve good scalable NUMA (nonuniform memory access) performance. This technology, and avoiding global data, global code paths, and global locks, allows K42's design to scale to thousands of processors.
- Built on K42's object-oriented structure, *clustered objects*[25] provide an infrastructure to implement scalable services with the degree of distribution transparent to the client. This also facilitates autonomic multiprocessor computing, as K42 can dynamically swap between uniprocessor and multiprocessor clustered objects.
- K42 is designed to run on 64-bit architectures and we have taken advantage of 64 bits to make performance gains by, for example, using large virtually sparse arrays rather than hash tables.
- K42 is fully preemptable and most of the kernel data structures are pageable.
- K42 is designed to support a simultaneous mix of time-shared, real-time, and fine-grained gang-scheduled applications.
- K42 has developed deferred object deletion[25] similar to RCU,[26] in which objects release their locks before calling other objects. This efficient programming model is crucial for multiprocessor performance and is similar to type-safe memory.[27]

***K42 overall status.*** K42 is available under an LGPL[28] license (a source tree is available at the URL in Reference 2). We are actively working on providing a complete environment including build and debug tools, simulator, and source. The modular structure of the system makes it a good teaching, research, and prototyping vehicle. Some of the policies and implementations studied in this framework have been transferred into "vanilla" Linux, and we expect that work to continue. Also, in the long term, we expect that the kind of base technologies we are exploring with K42 will be important to Linux.

K42 currently runs on 64-bit MIPS (NUMA) and PowerPC (SMP—symmetric multiprocessor) platforms and is being ported to x86-64. As stated, K42 is fully functional for 64-bit applications, and can run codes ranging from scientific applications to complex benchmarks like SDET, to significant subsystems like Apache. We have demonstrated better base performance for real applications and demonstrated better scalability than other commercial OSs. We expect in the near future to achieve full self-hosting and demonstrate that specialized subsystems can customize the OS to achieve better performance at reduced complexity. There are still edge conditions that have not yet been addressed, and there are still objects with only simplistic implementations.

**Infrastructure for hot swapping.** In K42, each virtual resource instance, for example, open file instance or memory region, is implemented by combining a set of (C++) object instances we call building blocks.[29] Each building block implements a particular abstraction or policy and might (1) manage some part of the virtual resource, (2) manage some of the physical resources backing the virtual resource, or (3) manage the flow of control through the building blocks. For example, there is no global page cache in K42; instead, for each file there is an independent object that caches the blocks of that file.

K42's infrastructure allows any object to replace any other object implementing the same interface, or to interpose any object with one providing the same interface. In K42 we use hot swapping to replace kernel objects as well as objects in user-level servers. The hot swapping occurs transparently to the clients of the component and no support or code changes are needed in the clients.

***Dynamic replacement algorithm: Design issues.*** This algorithm contains the following steps: (1) instantiate the replacement component, (2) establish a quiescent state for the component to be replaced, (3) transfer state from the old component to the new component, (4) swap in the new component replacing all references, and (5) deallocate the old component.

There are three key issues that need to be addressed in this design. First, we need to establish a quiescent state so that it is safe to transfer state and swap references. The swap can only be done when the component state is not currently being accessed by any thread in the system. Perhaps the most straightforward way to achieve a quiescent state would be to require all clients of the component to acquire a reader-writer lock in read mode before any call to the component. Acquiring this external lock in write mode would thus establish that the component is safe for swapping. However, this would add overhead for the common case, and cause locality problems in the case of multiprocessors. Further, the lock could not be part of the component itself.

Second, we need to determine what state needs to be transferred and how to transfer it to the new component safely and efficiently. Although the state could be converted to a canonical, serialized form, this would lose context, be a less efficient transfer protocol, and potentially prevent parallelism when the transfer is occurring on a multiprocessor.

Third, we need to swap all of the references held by the clients of the component so that the references refer to the new component. In a system built around a single, fully typed language, like the Java** language, this could be done using the same infrastructure as used by garbage collection systems. However, this would be prohibitively expensive for a single component switch. An alternative would be to partition a hot-swappable component into a front-end component and a back-end component, where the front-end component is referenced (and invoked) by the component clients, and is used only to forward requests to the back-end component. There would then be only a single reference (in the front-end component) to the back-end component that would need to be changed when a component is swapped, but this adds extra overhead to the common call path.

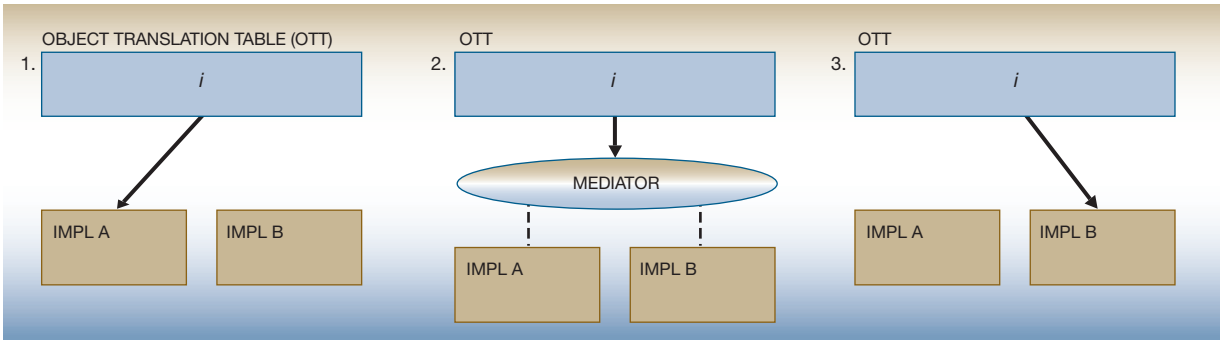
***K42 features used.*** Our implementation of hot swapping leverages three key features of K42 that allow us to address the issues listed above in an efficient and straightforward manner. Similar features exist or could be retrofitted into other systems.

First, because K42 has an object-oriented structure implemented using C++,[29] each system component maps naturally to a language object. Hot swapping is facilitated because the objects are self-contained with well-defined interfaces. A similar approach could be used in a system that is not object-oriented but uses operations tables, such as vnodes.[18]

Second, each K42 object is accessed through a single pointer indirection, where the indirection pointer of all objects is maintained in an Object Translation Table (OTT) indexed by the object identifier. The OTT was originally implemented in K42 to support a new form of scalable structure called *Clustered Objects*.[25] Another method for doing this would be the dynamic linking technology such as that used in ELF (Executable and Linking Format), otherwise the indirection would need to be added explicitly.

Finally, K42 has a *generation count* mechanism that allows us to easily determine when all threads that were started before a given point in time have completed, or reached a safe point. (A similar mechanism has been used by [formerly] Sequent's NUMA-Q**[30] for the same reason we originally developed it for K42, namely to improve multiprocessor performance by deferring expensive, but noncritical operations.[31]) This mechanism is used to achieve a quiescent state for an object. The mechanism exploits the fact that OSs are event-driven, where most

Figure 4 Stages in replacing instance A of object *i* by instance B




requests are serviced and completed quickly. Long-living daemon threads are treated specially. This type of functionality can usually be added to other event-driven systems, such as Web, file, or database servers, in which the thread of control frequently reaches safe points such as the completion of a system call, or entering a sleep.

***Description of algorithm.*** A part of the replacement algorithm involves interpositioning a mediator. Interpositioning is the ability to redirect future calls intended for a given object to another object. To perform interpositioning the object's indirection pointer in the OTT is changed to point to the interposed object. It is not necessary to reach object quiescence. This interposition object remains active and can forward calls to the original object performing whatever operation or monitoring it desires prior to the forwarding.

Conceptually there are three stages in replacing a component as depicted in Figure 4. In the OTT's initial state, the *i*-th table entry contains a pointer to the current (old) object. In the second stage, a mediator is interposed. In the third stage the OTT is in its final state, with the *i*-th entry pointing to the new object.

To perform a replacement, a mediator object is interposed in front of the old object. This mediator object proceeds through the three phases. To establish a quiescent state, in which it is guaranteed that no threads have an active reference to the component to be swapped, the mediator object initially, in the *Forward* phase, tracks all threads making calls to the component and forwards each call on to the original component. It does so until it is certain that all calls started before call tracking began have completed. To detect this, we rely on K42's generation count feature to determine when all calls that were started before tracking began have completed. Until that point, the mediator continues to forward new requests to the original component, which services them as normal.

At that point, the mediator starts the *Blocked* phase and temporarily blocks new calls, while it waits for the tracked calls to complete. Exceptions are made for recursive calls that need to be allowed to proceed to avoid deadlock. If the user subverts the K42 programming model making recursive object calls across servers while simultaneously switching multiple objects, the infrastructure will not be able to detect deadlock loops. To handle this, and potentially other unknown deadlock circumstances, a timeout mechanism is used. If the timeout is triggered, it terminates the hot swap, setting the client pointers back to the original object. In nonerroneous object implementations, calls can be correctly tracked and blocked. Once all the tracked calls have completed, the component is in a quiescent state, and the state transfer can begin. While the *Blocked* phase may seem to unduly reduce the responsiveness of the component, in practice the delay depends only on the number of tracked calls, which are generally short and few in number.

To make state transfer between the original and the new component efficient and preserve as much of the original state and semantics as possible, the original and new objects negotiate a *best common format* that they both support. This, for example, may allow a hash table to be passed directly through a pointer, rather than converted to and from some ca-

nonical form, such as a list or array, as well as, in a large multiprocessor, allow much of the transfer to occur in parallel across multiple processors, preserving locality where possible.

Finally, after the state transfer, the mediator enters the *Completed* phase. It removes its interception by updating the appropriate indirection pointer in the OTT to point to the new component so that future calls go to the new component directly. It also resumes all threads that were suspended during the *Blocked* phase and directs them to the new component. The mediator then deallocates the original object and finally itself.

The implementation of the above design has a number of important features. There is no overhead during normal operation; overhead occurs only when the mediator is interposed, and the mediator is used only during the hot-swapping process. The implementation works efficiently in a multithreaded multiprocessor system. The mediator runs in parallel and state transfer proceeds in parallel. Call interception and mediation is transparent to the clients, facilitated by K42's component system infrastructure. Finally, our solution is generic in that it separates the complexity of swap-time in-flight call tracking and deadlock avoidance from the implementation of the component itself. With the exception of component state transfer, the rest of the swapping process does not require support from the component, simplifying the addition of components that wish to take advantage of the hot-swapping capability.

**Status, performance, and continuing work.** K42 fully supports the hot-swapping infrastructure described in the previous section. We have used it for simple applications, such as the search program in Figure 1, as well as for complex applications, such as the SPEC SDET benchmark. The code works for all the objects in our system, and the main current limitation is in the number of choices of objects we can hot swap. We are continuing to explore the use of hot swapping in implementing additional autonomic computing features. Next we present a performance study involving a significant application, and then we discuss our plans for the future.

An important aspect of the virtual memory system is keeping track of in-core pages. In K42, this function is implemented by a File Cache Manager (FCM) component. We focus on two of the default implementations: shared and distributed. For each open file, an instance of an FCM is used to cache the pages of the file's data in physical frames. By default, to achieve better performance when a file is opened, a simple shared implementation of the FCM is created. The default decision is made based on the fact that most files are accessed by one thread on one processor and opened only briefly. If the file is accessed on one processor, the shared FCM implementation performs well and has little memory overhead. If the file is accessed by multiple processors concurrently, the associated FCM is hot swapped to a distributed implementation. This alleviates contention and yields better scalability, thus ensuring that only the files that experience contention due to sharing use the more complex and expensive distributed FCM implementation. However, because the shared implementation performs an order of magnitude worse when running on many processors, without hot swapping, an FCM suitable for the distributed case would need to be used all the time and a performance penalty paid in the single processor case.

One of the studies we did to understand the advantages of hot swapping the FCM implementations was to run the PostMark[32] and SPEC SDET benchmarks. PostMark is a uniprocessor file system benchmark that creates a large number of small files on which a number of operations are performed, including reading and appending. SPEC SDET is a multiprocessor UNIX development workload that simulates concurrent users running standard UNIX commands (Due to tool chain issues we used a modified version that does not include a compiler or UNIX command "ps."). If we disable hot swapping and run PostMark using only shared FCMs, and then run it using only distributed FCMs, we find that we suffer a 7 percent drop in performance for the distributed implementation of the FCM. On the other hand, if we run SDET in a similar fashion we find that the distributed FCM gets an 8 percent performance improvement on 4 processors and an order of magnitude improvement on 24 processors over the shared FCM. When hot swapping is enabled, the best performance is achieved automatically for both PostMark and SDET, with the individual FCM instances choosing the right implementation based on the demands it experiences.

The above experiment shows the power of hot swapping. In other work we have studied the performance advantages of hot swapping. The results we have obtained are encouraging, but there is still much to do within the K42 project to bring hot swapping to a mature state. Currently the trigger for an object hot swap is either specified by an object or by an appli-

cation via an explicit request. Although this has proven to be sufficient for many cases, we have found situations where a monitoring infrastructure would allow us to make the trigger decision on behalf of an object. The monitoring code we currently use has often been placed in the object for convenience and to gain experience with how to gather and use the information. We plan to use object interposition to reduce the overhead of this scheme and provide a more generic mechanism for gathering this information.

Determining the target object is done today explicitly by the requester of a hot swap. We are currently extending the K42 type system to provide a service that identifies objects by their characteristics, again allowing a more generic mechanism for handling the hot swap. Because our project has been performance-driven, our focus so far has been to use autonomic computing to improve performance. However, there are important benefits to be gained from being able to autonomically swap in a security patch, or achieve higher availability by a live upgrade. Thus, we are beginning to explore other autonomic features.

There is considerable potential, and required effort, to understand the longer-term and larger issues involved in providing many components that manage a given resource and getting them to safely and correctly interact. So far, we have successfully used hot swapping and its autonomic features in K42 and expect continued progress in our research.

## Related work

Although there is a large body of prior work focusing on the downloading and dynamic binding of new components, there has been less work on swapping components in a live system. Hjálmtýsson and Gray describe a mechanism for updating C++ objects in a running program,[21] but their client objects need to be able to recover from broken bindings due to an object swap and retry the operation, so their mechanism is not transparent to client objects. Moreover, they do not detect quiescent state, and old objects continue to service prior calls while the new object begins to service new calls.

The *virtualizing Operating System* (vOS)[33] is a middleware application that offers an abstraction layer between applications and the underlying OS. vOS allows modules to be refreshed, that is, to be dynamically reloaded to achieve software rejuvenation. It does not address loading of new implementations, and its approach does not apply to OS components.

Pu et al. describe a "replugging mechanism" for incremental and optimistic specialization,[17] but they assume there can be at most one thread executing in a swappable module at a time. In later work that constraint is relaxed but is nonscalable. Hicks et al. describe a method for dynamic software updating, but in their approach, all objects of a certain type are updated simultaneously, not just individual instances, as is possible with our scheme.[20] Moreover, they require that the program be coded to decide when a safe point has been reached and initiate the update.

The modular structure, level of indirection, and availability of a mechanism for detecting a quiescent state is not unique to K42. Sequent has a mechanism for detecting quiescent state,[31] and we are working with this group to incorporate a similar facility into Linux. This will allow hot swapping of system modules (i.e., device drivers and file systems).

In general, the work described here can be viewed as part of wide-spread research efforts to make OSs more adaptive and extensible as in SPIN,[10] Exokernel,[11] and VINO.[13] These systems are unable to swap entire components, but rather just provide hooks for customization. Our work is complementary to the above mentioned related work. We focus primarily on a mechanism for swapping generic components in a highly dynamic, multithreaded multiprocessing system. Several groups have also done work on adding extensibility to both applications and systems. CORBA**,[34] DCE,[35] and RMI[36] are all application architectures that allow components to be modified during program execution; but they do not address the performance or complexity concerns present in an OS.

## Conclusions

Autonomic systems software should have self-maintenance capabilities, and should be able to run applications well on various hardware platforms and across various environments. We proposed an object-oriented system structure that supports hot swapping as an infrastructure toward meeting these autonomic challenges by dynamically monitoring and modifying the OS. We demonstrated the technical feasibility of the hot-swapping approach by describing our implementation of it in K42. We described how hot swapping can yield performance advantages and

showed an application involving K42. We will continue to explore other autonomic features in K42 by examining security, software upgrading, and multiple object coordination issues.

Our prototype is Open Source Software and available on our Web site.[2] Readers may use the Web site for accessing related white papers, and for contacting us if interested in participating in this project.

## Acknowledgments

We would like to thank the anonymous referees for their positive and helpful comments.

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of The Open Group, Linus Torvalds, MIPS Technologies, Inc., Sun Microsystems, Inc., Sequent Computer Systems, Inc., or Object Management Group.

## Cited references

1. IBM Corporation, Autonomic Computing, http://www.research.ibm.com/autonomic/.
2. IBM Corporation, Research Division, The K42 Project, http://www.research.ibm.com/K42.
3. D. Knebel et al., "Diagnosis and Characterization of Timing-Related Defects by Time-Dependent Light Emission," *Proceedings of the International Test Conference*, October 18–23, 1998, Washington, DC, IEEE, New York (1998), pp. 733–739.
4. M. Paniccia, T. Eiles, V. R. M. Rao, and W. M. Yee, "Novel Optical Probing Technique for Flip Chip Packaged Microprocessors," *Proceedings of the International Test Conference*, October 18–23, 1998, Washington, DC, IEEE, New York (1998), pp. 740–747.
5. O. Krieger, M. Stumm, and R. Unrau, "The Alloc Stream Facility: A Redesign of Application-Level Stream I/O," *IEEE Computer* **27**, No. 3, 75–82 (1994).
6. O. Krieger and M. Stumm, "HFS: A Performance-Oriented Flexible File System Based on Building-Block Compositions," *ACM Transactions on Computer Systems* **15**, No. 3, 286–321 (1997).
7. Reiser File System (ReiserFS), see http://www.namesys.com.
8. G. Glass and P. Cao, "Adaptive Page Replacement Based on Memory Reference Behavior," *Proceedings of the 1997 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, Seattle, WA, ACM Press, New York (1997), pp. 115–126.
9. J. M. Kim, J. Choi, J. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim, "A Low-Overhead High-Performance Unified Buffer Management Scheme That Exploits Sequential and Looping References," *Proceedings, Symposium on Operating Systems Design and Implementation*, San Diego, CA, USENIX Association (2000), pp. 119–134.
10. B. N. Bershad, S. Savage, P. Pardyn, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers, "Extensibility, Safety and Performance in the SPIN Operating System," *Proceedings, ACM Symposium on Operating System Principles*, Copper Mountain Resort, CO, ACM, New York (1995), pp. 267–283.
11. D. R. Engler, M. F. Kaashoek, and J. O'Toole, "Exokernel: An Operating System Architecture for Application-Level Resource Management," *Proceedings, ACM Symposium on Operating System Principles*, Copper Mountain Resort, CO, ACM, New York (1995), pp. 251–266.
12. D. Mosberger and L. L. Peterson, "Making Paths Explicit in the Scout Operating System," *Proceedings, Symposium on Operating Systems Design and Implementation*, Seattle, WA, ACM, New York (1996), pp. 153–167.
13. M. Seltzer, Y. Endo, C. Small, and K. A. Smith, *An Introduction to the Architecture of the VINO Kernel*, Technical Report TR-34-94, Harvard University, Cambridge, MA (1994).
14. G. R. Ganger, D. R. Engler, M. F. Kaashoek, H. M. Briceno, R. Hunt, and T. Pinckney, "Fast and Flexible Application-Level Networking on Exokernel Systems," *ACM Transactions on Computer Systems* **20**, No. 1, 49–83 (2002).
15. A. D. Brown, T. C. Mowry, and O. Krieger, "Compiler-Based I/O Prefetching for Out-of-Core Applications," *ACM Transactions on Computer Systems* **19**, No. 2, 111–170 (2001).
16. R. V. Meter and M. Gao, "Latency Management in Storage Systems," *Proceedings, Symposium on Operating Systems Design and Implementation*, San Diego, CA, USENIX Association (2000), pp. 103–117.
17. C. Pu, T. Autrey, A. Black, C. Consel, C. Cowan, J. Inouye, L. Kethana, J. Walpole, and K. Zhang, "Optimistic Incremental Specialization: Streamlining a Commercial Operating System," *Proceedings, ACM Symposium on Operating System Principles*, Copper Mountain Resort, CO, ACM, New York (1995), pp. 314–321.
18. S. R. Kleiman, "Vnodes: An Architecture for Multiple File System Types in Sun UNIX," *Proceedings of the Summer Conference*, Atlanta, GA, 1986, USENIX Association (1986), pp. 238–247.
19. R. Gooch, "Linux Virtual File System," http://www.atnf.csiro.au/people/rgooch/linux/vfs.txt.
20. M. Hicks, J. T. Moore, and S. Nettles, "Dynamic Software Updating," *Proceedings of the ACM SIGPLAN'01 Conference on Programming Language Design and Implementation*, Snowbird, UT, ACM Press, New York (2001), pp. 13–23.
21. G. Hjálmtýsson and R. Gray, "Dynamic C++ Classes: A Lightweight Mechanism to Update Code in a Running Program," *Proceedings, Annual USENIX Technical Conference*, USENIX Association (1998), pp. 65–76.
22. J. Aman, C. K. Eilert, D. Emmes, P. Yocom, and D. Dillenberger, "Adaptive Algorithms for Managing a Distributed Data Processing Workload," *IBM Systems Journal* **36**, No. 2, 242–283 (1997).
23. Y. Li, S.-M. Tan, Z. Chen, and R. H. Campbell, *Disk Scheduling with Dynamic Request Priorities*, Technical Report, University of Illinois at Urbana-Champaign, IL (August 1995).
24. J. Liedtke, "On Micro-Kernel Construction," *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, Copper Mountain, CO, ACM Press, New York (1995), pp. 237–250.
25. B. Gamsa, O. Krieger, J. Appavoo, and M. Stumm, "Tornado: Maximizing Locality and Concurrency in a Shared Memory Multiprocessor Operating System," *Proceedings, Symposium on Operating Systems Design and Implementation*, New Orleans, LA, USENIX Association (1999), pp. 87–100.
26. P. E. McKenney, D. Sarma, A. Arcangeli, A. Kleen, O. Krieger, and R. Russell, "Read Copy Update," *Proceedings of the Ottawa Linux Symposium*, Ottawa, Canada (June 2002).

27. M. Greenwald and D. Cheriton, "The Synergy Between Non-Blocking Synchronization and Operating System Structure," *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation*, Seattle, WA, ACM Press, New York (1996), pp. 123–136.
28. Free Software Foundation, Inc., GNU Lesser General Public License, http://www.research.ibm.com/K42/LICENSE.html.
29. M. Auslander, H. Franke, B. Gamsa, O. Krieger, and M. Stumm, "Customization Lite," *Hot Topics in Operating Systems*, IEEE, New York (May 1997), pp. 43–48.
30. T. Lovett and R. Clapp, "STiNG: A CC-NUMA Computer System for the Commercial Marketplace," *Proceedings of the 23rd International Symposium on Computer Architecture*, ACM, New York (1996), pp. 308–317.
31. P. E. McKenney and J. D. Slingwine, "Read-Copy Update: Using Execution History to Solve Concurrency Problems," International Conference on Parallel and Distributed Computing and Systems, Las Vegas, NV, 28–31 October 1998, IASTED (International Association of Science and Technology for Development), Calgary, Canada (1998).
32. J. Katcher, *PostMark: A New File System Benchmark*, TR3022, Network Appliance, Sunnyvale, CA 94089, http://www.netapp.com/tech_library/3022.html.
33. T. Boyd and P. Dasgupta, "Preemptive Module Replacement Using the Virtualizing Operating System," Workshop on Self-Healing, Adaptive and Self-Managed Systems (SHAMAN) (June 2002).
34. Z. Yang and K. Duddy, "CORBA: A Platform for Distributed Object Computing," *ACM Operating Systems Review* **30**, No. 2, 4–31 (1996).
35. *Distributed Computing Environment: Overview*, OSF-DCE-PD-590-1, Open Software Foundation (May 1990).
36. Java Remote Method Invocation—Distributed Computing for Java, Sun Microsystems, Inc., http://java.sun.com/marketing/collateral/javarmi.html.

*Accepted for publication July 29, 2002*

**Jonathan Appavoo** *Department of Computer Science, University of Toronto, 10 King's College Road, Toronto, Ontario, Canada M5S 3G4 (electronic mail: jonathan@eecg.toronto.edu).* Mr. Appavoo has a Master of Computer Science degree from the University of Toronto, where he is currently enrolled in the Ph.D. program. He has spent the last four years working on K42 and its predecessor, Tornado. He works closely with the K42 team at the IBM Thomas J. Watson Research Center, where he has interned. His research has focused on multiprocessor operating system performance with a particular interest in scalable data structures.

**Kevin Hui** *Kaleidescape, Inc., 155 Frobisher Drive, Suite I-205, Waterloo, Ontario, Canada N2V 2E1 (electronic mail: kevin@kaleidescape.com).* Mr. Hui is a software engineer at Kaleidescape, where he works on operating system and information security. Under the supervision of Dr. Michael Stumm, he obtained his M.Sc. in computer science in 2000 at the University of Toronto. Prior to receiving his degree, he enjoyed a productive stay at the IBM Thomas J. Watson Research Center, where he designed and implemented K42's hot-swapping infrastructure with Dr. Robert Wisniewski and the rest of the K42 team.

**Craig A. N. Soules** *Computer Science Department, Carnegie Mellon University, Pittsburgh, Pennsylvania 15212 (electronic mail: soules@cmu.edu).* Mr. Soules has been a graduate student in the Computer Science Department at Carnegie Mellon University since fall 2000. His research has focused mainly on the design and performance of operating systems and file systems.

**Robert W. Wisniewski** *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (electronic mail: bob@watson.ibm.com).* Dr. Wisniewski is a research scientist at the Watson Research Center working on the K42 operating system, and is currently exploring scalable, portable, and configurable next-generation operating systems. He received his Ph.D. degree in 1996 from the University of Rochester, Rochester, NY, where his thesis was "Achieving High Performance in Parallel Applications via Kernel-Application Interaction." His research interests include scalable parallel systems, first-class system customization, and performance monitoring.

**Dilma M. Da Silva** *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (electronic mail: dilmasilva@us.ibm.com).* Dr. Da Silva received her B.S. and M.S. degrees in computer science from the University of Sao Paulo, Brazil, and her Ph.D. from Georgia Institute of Technology, Atlanta, GA. From 1996 to 2000 she was an assistant professor in the Department of Computer Science at the University of Sao Paulo, Brazil. She is currently a research staff member at IBM's Thomas J. Watson Research Center. Her research interests include operating systems and dynamic system configuration.

**Orran Krieger** *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (electronic mail: okrieg@us.ibm.com).* Dr. Krieger is a manager at IBM's Thomas J. Watson Research Center. He received a B.A.Sc. degree from the University of Ottawa in 1985, an M.A.Sc. degree from the University of Toronto in 1989, and a Ph.D. degree from the University of Toronto in 1994, all in electrical and computer engineering. He was one of the main architects and developers of the Hurricane and Tornado operating systems at the University of Toronto, and was heavily involved in the architecture and development of the Hector and NUMAchine shared-memory multiprocessors. Currently, he is project leader of the K42 operating system project at the IBM Research Center and an adjunct associate professor in computer science at Carnegie Mellon University, Pittsburgh, PA. His research interests include operating systems, file systems, and computer architecture.

**Marc A. Auslander** *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (electronic mail: Marc_Auslander@us.ibm.com).* Mr. Auslander is an IBM Fellow, a member of the National Academy of Engineering, an ACM Fellow, and an IEEE Fellow. He received the A.B. in mathematics from Princeton University in 1963. He joined IBM in 1964 and transferred to the Thomas J. Watson Research Center in 1968. Mr. Auslander contributed significantly to IBM's entry into virtual systems. He was an original member of the group that designed and built the first IBM RISC machine and the first RISC optimizing compiler. He has also made important contributions to AIX®, other operating systems, and the PowerPC architecture. He is currently working on the K42 operating system. His research interests include multiprocessor operating systems, compilers, and processor architecture.

**David J. Edelsohn** *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (electronic mail: dje@watson.ibm.com).* Dr. Edelsohn is a research

staff member in IBM's Research Division. He received his Ph.D. degree in 1996 from Syracuse University, Syracuse, NY. At Syracuse University's Northeast Parallel Architectures Center, he worked with Geoffrey Fox studying mixed media systems and massively parallel hierarchical algorithms. His research interests include compiler optimizations, operating systems, and parallel computing.

**Ben Gamsa** *SOMA Networks, Inc., 312 Adelaide Street West, Suite 700, Toronto, Ontario, Canada M5V 1R2 (electronic mail: ben@somanetworks.com).* Dr. Gamsa received his Ph.D. degree in 1999 from the Department of Computer Science at the University of Toronto and is currently employed at SOMA Networks.

**Greg R. Ganger** *Carnegie Mellon University, Pittsburgh, Pennsylvania 15212 (electronic mail: greg.ganger@cmu.edu).* Dr. Ganger is a professor in the Electrical and Computer Engineering Department at Carnegie Mellon University, Pittsburgh, PA. His broad research interests in computer systems include storage systems, security, and operating systems. He is director of Carnegie Mellon's Parallel Data Lab, academia's premiere storage systems research center. His Ph.D. in computer science and engineering is from the University of Michigan, and he spent two and one half years as a postdoctoral student at Massachusetts Institute of Technology, working on the Exokernel project.

**Paul McKenney** *IBM Server Group, Linux Technology Center, 15450 S.W. Koll Parkway, Beaverton, Oregon 97006-6096 (electronic mail: pmckenne@us.ibm.com).* Mr. McKenney is a system architect at the Linux Technology Center. Prior to that he worked on locking and parallel operating-system algorithms at Sequent Computer Systems. He is working toward his Ph.D. degree at Oregon Graduate Institute. His research interests include parallelism, virtualization, and autonomic computing.

**Michal Ostrowski** *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (electronic mail: mostrows@watson.ibm.com).* Mr. Ostrowski is a member of the Advanced Operating Systems (K42) group at the IBM Research Center. He is involved in research and development activities on the K42 operating system, currently focusing on developing a framework for using Linux kernel code in K42 and developing frameworks and mechanisms for efficient asynchronous I/O interfaces. He completed his master's degree at the University of Waterloo in 2000.

**Bryan Rosenburg** *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (electronic mail: rosnbrg@us.ibm.com).* Dr. Rosenburg received his Ph.D. degree in computer sciences at the University of Wisconsin-Madison in 1986, and immediately thereafter joined the Watson Research Center as a research staff member in the operating system group of the RP3 NUMA multiprocessor project. He has been a member of the K42 team since its inception. His current research interests are in the lowest levels of operating system architecture: interrupt handling, context switching, interprocess communication, and low-level scheduling.

**Michael Stumm** *Department of Computer Science, University of Toronto, 10 King's College Road, Toronto, Ontario, Canada M5S 3G4 (electronic mail: stumm@eecg.toronto.edu).* Dr. Stumm is a professor in the Department of Electrical and Computer Engineering and the Department of Computer Science at the University of Toronto. He received a diploma in mathematics and a Ph.D. degree in computer science from the University of Zurich in 1980 and 1984, respectively. Dr. Stumm's research interests are in the areas of computer systems, in particular operating systems for distributed and parallel systems.

**Jimi Xenidis** *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (electronic mail: jimix@watson.ibm.com).* Mr. Xenidis is a software engineer at the Watson Research Center working with several groups in addition to the K42 project. His research interests include operating systems, linkers and libraries, and machine virtualization.

# Toward a new landscape of systems management in an autonomic computing environment

by G. Lanfranchi
P. Della Peruta
A. Perrone
D. Calvanese

In this paper we present IBM Tivoli Monitoring, a systems management application that displays autonomic behavior at run time, and we focus on extending it in order to encompass the design and the deployment phases of the product life cycle. We review the resource model concept, illustrate it with an example, and discuss its role throughout the product life cycle. Then we introduce basic concepts in ontology and description logics and discuss representing Common Information Model constructs using description logics. Finally, we propose Systems Management Ontology, an approach to enhancing the autonomic properties of IBM Tivoli Monitoring based on an ontology service and the technique of "contextual pulling" applied to the resource model.

*Problems cannot be solved at the same level of awareness that created them.*

—Albert Einstein

The information technology (IT) infrastructure that supports business systems today continues to evolve at a breakneck pace, with the integration of new devices, servers, and applications creating highly complex systems. Systems management needs to similarly evolve in order to cope with this increasing complexity in IT. Management tools need to become "smarter" if they are to drive successful business systems. Performance and availability management tools are, in particular, key enablers of an efficient and profitable business. A business cannot execute efficiently unless the mission-critical business applications and the supporting middleware and operating systems are available and performing well. Any component failure or poor performance could adversely impact the business.

To minimize the business downtime, the speedy execution of the appropriate corrective action is needed. An important component of performance and availability management is the monitoring of the system in order to detect anomalies as soon as they occur and to take the necessary corrective actions (e.g., restore the failing objects to their desired state, or activate backup resources). The monitoring and the taking of corrective action should be optimized with respect to the overall objectives of the entire business system. Whenever feasible, systems management tools with predictive capabilities should be used in order to detect future problem states, and to allow action to be taken before a failure occurs and adversely impacts users.

When the tools monitoring the system resources simply collect raw data (e.g., performance metrics), it can be difficult to draw any conclusion about the health of a system. A graphical performance monitor or an event viewer tool can be useful, but only if a skilled administrator is able to interpret the information provided, and to determine if a problem exists or not. Only an experienced administrator has the ability to correlate the appropriate domain

knowledge with the data collected, such as metrics and events, and come up with answers to questions such as: Is there a memory bottleneck?, What is its cause?, and How can I fix it? In the current monitoring environment, the system administrator inherently owns the best practices and applies them in order to identify and cure problems.

We describe here the approach taken in IBM Tivoli Monitoring,[1] in which the monitoring tool directly models and implements the relevant aspects of the domain entities as logical objects, and via this implementation it transforms the raw data into the information required in order to detect, correct, and, whenever possible, predict abnormal system behavior. The system administrator is thus better able to address IT problems quickly, and thus concentrate on better serving business demands.

The best practices embedded in the monitoring tool enable the autonomic behavior of the monitoring tool at run time.[2] This autonomic behavior can be extended to the other two phases of the application life cycle: the design phase (where the best practices are created), and the deployment phase (where the best practices are deployed into the managed systems).

In this paper we first present IBM Tivoli Monitoring as an existing solution that displays autonomic behavior at run time, and then we focus on extending this solution to encompass the design time and the deployment time. In the next section, we provide an overview of IBM Tivoli Monitoring with a particular focus on the "resource model" concept. In the following section we present the Systems Management Ontology with an overview of "description logics" and their use in representing Common Information Model constructs. Finally, we link the previous themes in order to illustrate the proposed approach for autonomic systems management.

## The IBM Tivoli Monitoring solution

The IBM Tivoli Monitoring (ITM) solution is based on the resource model concept. In this section we define the resource model and illustrate it through an example. We then discuss the application of the resource model to all phases of the ITM life cycle.

**The resource model concept.** The resource model is the main tool for implementing an "identify, notify, and cure" systems management strategy. A resource model has two parts: a dynamic model and a reference model.

The Common Information Model (CIM) formalism promoted by the Distributed Management Task Force is a way to organize the available information about the managed environment that applies the basic structuring and conceptualization techniques of the object-oriented paradigm.[3] The approach uses a uniform modeling formalism that, together with the basic repertoire of object-oriented constructs, supports the cooperative development of an object-oriented schema across multiple organizations. The *dynamic model* uses CIM classes and CIM properties to describe resources (such as memory) and their performance metrics (such as process working set). Moreover, it uses CIM methods to describe actions that can be executed against the resource (such as starting a process). The CIM association is then used to represent the resource within a high level object (logical object).

The *reference model*, implemented as a decision procedure and coded using either Visual Basic** or JavaScript**, incorporates best practices that:

- Interpret the status of a problem object against a defined baseline using the metrics provided by the dynamic model
- Discover the root cause of the problem
- Correct the problem
- Log performance data related to the domain objects

The reference model describes the "critical paths" within systems or applications. It interprets the "quality" of applications and systems against a predefined service level in order to discover the root cause of the problem and to react accordingly. The reference model has a direct link to the dynamic model describing the resources and it analyzes the properties of objects aggregated in the model, generates indications about the status of those resources, and invokes methods to correct a problem. The reference model, in effect, implements the best practices normally used by a system administrator to detect problems and to identify their root cause.

A resource model is created for each problem and it contains the best practices used to identify and correct a well-defined problem. Figure 1 contains an example of a resource model used to identify memory problems in a Microsoft Windows** machine. The dynamic model is created using a CIM representa-

Figure 1 Resource model for detecting memory problems in Microsoft Windows machines

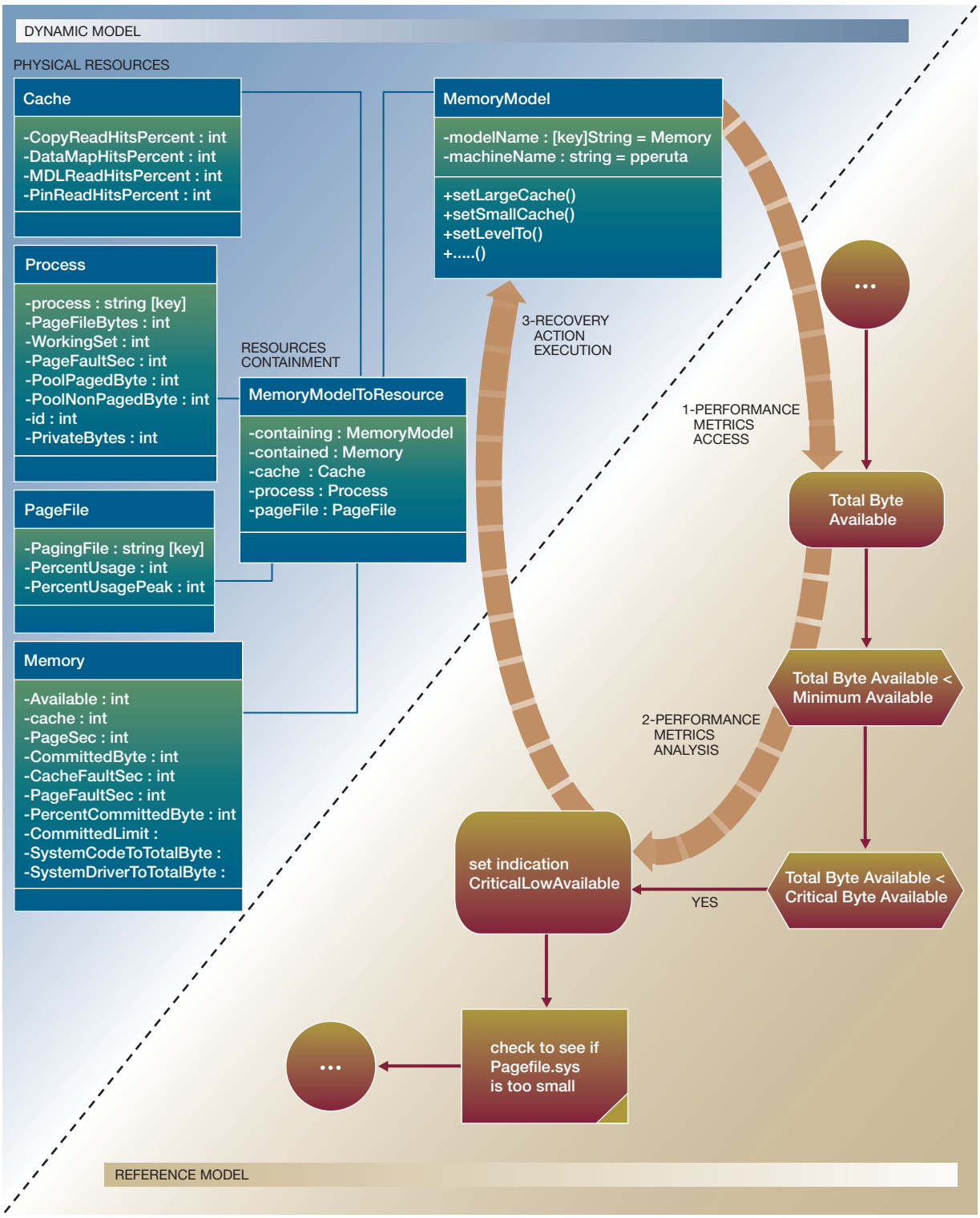
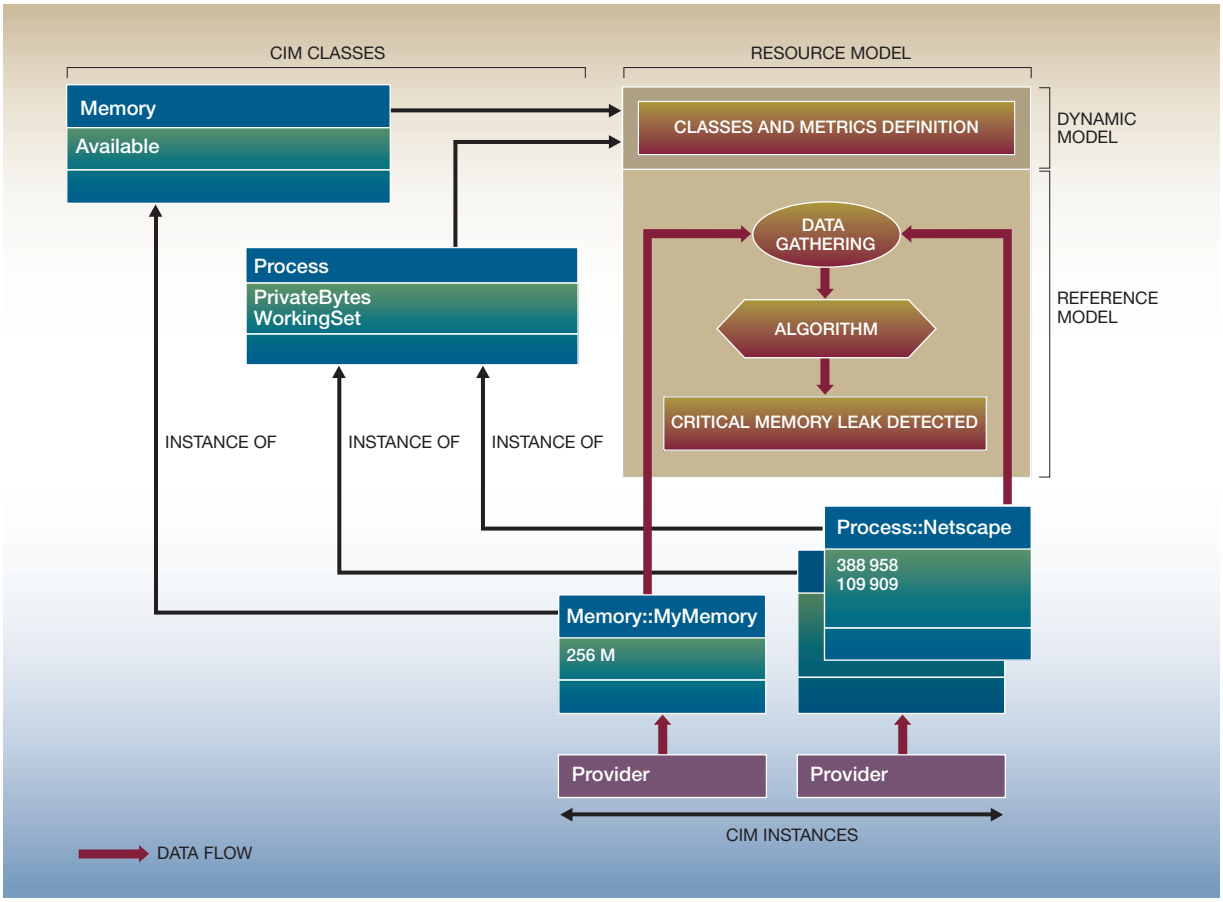


Figure 2 A simplified resource model for detecting critical memory leaks




tion of the resources (Cache, Process, PageFile, Memory) and the properties that are useful in detecting memory problems. These classes are then associated via association class MemoryModelToResource with object MemoryModel. Class MemoryModel also contains a set of methods, such as setLargeCache, that can be used to modify the status of some resource (e.g., Cache).

The reference model is linked to class MemoryModel (and thus to the dynamic model) and, using this class, it can access the performance metrics of the associated resources and apply the encoded best practices in order to discover if a memory problem exists. When a problem is detected, the reference model can invoke a well-defined method in MemoryModel to fix the problem, and also to notify the system operator about the problem resolution. The best practices are implemented in the form of an algorithm that analyzes and correlates metrics, and compares the results against a well-defined baseline.

Let us analyze a simplified resource model for detecting critical memory leaks (see Figure 2). Classes Memory and Process, as well as other additional classes not shown in Figure 2, constitute the dynamic model. MyMemory is an instance of class Memory, whereas Netscape is an instance of class Process. Notice that property Available of class Memory, and properties PrivateBytes and WorkingSet of class Process are represented in class instances MyMemory and Netscape by their numerical values, respectively 256, 388958, and 109909. The function of layer Provider is to gather the raw metrics (from the actual resources) in order to supply the above-mentioned values for properties. The reference model controls the

data gathering and interprets the data according to best-practices algorithms in order to detect the critical "memory leak" condition.

A "critical memory leak" occurs when the conditions "low available memory with high working set" and "memory leak in private bytes" exist. The condition "low available memory with high working set" can be detected by examining the number of bytes used for the working set of the process (property bytTotalWorkingSet), the number of bytes used as cache (bytTotalCache) and the total memory available (bytTotalAvail). Those three values can be calculated using "raw" metrics as follows.

- List all the process working sets and store the high working set in bytTotalWorkingSet
- Store the metric Cache from Memory object in bytTotalCache
- Store the metric Available from the Memory object in bytTotalAvail

The values collected for system resources are further processed in order to detect the "low available memory with high working set" condition.

```
bytTotalRAM = bytTotalWorkingSet +
  bytTotalCache + bytTotalAvail
numPercentWS =
  (bytTotalWorkingSet/bytTotalRAM) * 100
numPercentCache =
  (bytTotalCache/bytTotalRAM) * 100
numPercentAvail =
  (bytTotalAvail/bytTotalRAM) * 100
```

If (numPercentWS > numPercentCache and numPercentWS > numPercentAvail) then the condition "low available memory with high working set" is satisfied.

The condition "memory leak in private bytes" is generated from a metric of the Process resource PrivateBytes and observing its growth over time.

Store the process private bytes in PrivateBytes.

If the current value of process private bytes > the previous value of the process private bytes then the condition "memory leak in private bytes" is satisifed.

As shown, modeling the "critical memory leak" condition requires metrics provided by the CIM classes Memory and Process, and the above algorithm.

**The design, deploy, and run life-cycle phases.** The complete life cycle of the application, and of the resource model, consists of three phases: the design phase, the deployment phase, and the run-time (or operational) phase.

During the design phase, the resource model is designed, built, and tested. IBM Tivoli Monitoring supports this phase by providing an integrated development environment known as the Workbench. This is the phase when best practices are implemented and the domain knowledge is represented in the dynamic model and the reference model. Workbench users (developers, administrators, etc.) select the CIM classes for the dynamic model and then code the algorithms that analyze the resource metrics in order to detect and fix any abnormal behavior.

At deployment time, the resource model produced using the Workbench is installed in the management server from where it can be deployed to the target machines using the profile managers of the Tivoli Management Environment.[4,5]

On the target machine the resource model runs within ITM. Here, instances of classes included in the dynamic model are located, and the values of the properties for each of these objects are analyzed by the reference model. Abnormal conditions and performance degradation are detected and corrected. The information produced by the resource model and the performance metrics collected can be retrieved and analyzed using the ITM Web Health Console.[1]

**The next step.** Although the operational phase of monitoring displays autonomic properties—the resource model is able to keep resources under control with reduced human intervention—the design and deployment phases still require human intervention. Work is under way in the two following aspects to implement a more autonomic approach.

1. Apply automated reasoning procedures to the building of resource models. These procedures would help identify common knowledge, avoid redundancy, and optimize the set of resource models needed to manage a system or an application.
2. Provide a capability of the managed resource to "contextually pull" an appropriate set of resource models needed for its control. This will allow the management agent itself to pull the resource models from the central server, thus avoiding the need for the administrator to configure each monitor-

ing agent according to the exact set of applications present on the machine, and to push to the target the right set of resource models.

The next section highlights our blueprint for addressing these two objectives.

## Systems Management Ontology

In this section we introduce basic concepts in ontology and description logics in the context of Systems Management Ontology (SMO) and discuss representing CIM models in description logics. We use the memory model example to highlight the application of description logics reasoning capabilities to resource models.

**Ontologies and description logics.** *Ontologies* are metadata schemas, providing a controlled vocabulary of terms, each with an explicitly defined and machine understandable semantic.[6] They provide a shared and common understanding of a domain that can be communicated to people and intelligent systems, and thus facilitate knowledge sharing and reuse. Ontologies are intended to overcome the problem of implicit and hidden knowledge by making the conceptualization of a domain explicit. Researchers in artificial intelligence advocate expressing ontologies in logical terms, which makes them suitable for automated reasoning. Thus, information sharing and reuse are facilitated not only by the explicit representation of knowledge provided by ontologies, but also by the automated reasoning services that allow one to infer consequences from the available knowledge.

Nowadays there is consensus on structuring knowledge on a domain of interest in terms of classes of objects and relationships between classes. State-of-the-art ontology languages such as OIL and DAML+OIL[7] follow exactly this point of view. The need to model domains in terms of classes and relationships and the use of automated reasoning on such representations leads to *description logics* being offered as an ideal candidate for capturing ontologies on a given domain in logical terms. Indeed, both OIL and DAML+OIL can be viewed formally as dialects of description logics.

Description logics were introduced in the early 1980s in the attempt to provide a formal foundation to semantic networks and frame systems. Since then they have evolved into knowledge representation languages that are able to capture virtually all class-based representation formalisms used in artificial intelligence, software engineering, and databases.[8] One of the distinguishing features of the work on these logics is the detailed computational complexity analysis, both of the associated reasoning algorithms and of the logical implication problem that the algorithms are supposed to solve. Practical systems implementing such algorithms are now used in several projects.[9] Generally speaking, in description logics, the domain of interest is modeled by means of *concepts* (unary predicates) and *relations* (n-ary predicates), which denote classes of objects and relationships, respectively. Description logics are characterized by three basic components.

1. A *description language*, which specifies how to construct complex concept and relation expressions, by starting from a set of atomic concepts and relations and by applying suitable constructs
2. A *knowledge specification mechanism*, which specifies how to construct a description logic *knowledge base*, in which properties of concepts and relations are specified by means of suitable *assertions*
3. A set of *reasoning procedures*, which allow one to carry out suitable inferences from the knowledge expressed in the knowledge base

One of the most expressive description logics studied so far is DLR.[10–12] DLR is equipped with most of the concepts and relation constructs typical of decidable description logics (concrete domains are a notable exception) and allows for the most general form of knowledge assertions while still admitting decidable reasoning procedures.

**Representing CIM in description logics.** DLR has been successfully applied to representing UML (Unified Modeling Language) class diagrams[13] and to reasoning with them. Intuitively, classes are represented by atomic concepts, attributes, and aggregations by atomic (binary) relations, and each association of arity $n$ by an atomic concept and $n$ atomic binary relations. The latter allows for dealing with associations, with and without a corresponding association class, in a uniform way. The semantics of the various UML constructs is captured by suitable DLR assertions. We refer the reader to Reference 14 for the details of the encoding, while pointing out that the expressive power of DLR allows one to formalize several types of constraints that produce a better representation of the application semantics and that are typically not dealt with in a formal way. By making use of the encoding of UML class diagrams in DLR,

the following reasoning activities can be reduced to DLR reasoning tasks.[15,16]

- Consistency of the class diagram, that is, whether the diagram admits an instantiation. If this is not the case, there is no set of instances that satisfies the diagram, which indicates that the definitions altogether are inconsistent.
- Class consistency (association/aggregation consistency), that is, whether there is an instantiation of the diagram such that a given class (association/aggregation) has a nonempty extension. Observe that, if this is not the case, then there is an inconsistency in the class specification, or at the very least the class is inappropriately named since it is a synonym for the empty class.
- Class subsumption (association/aggregation subsumption), that is, whether the extension of one class (association/aggregation) is a subset of the extension of another class (association/aggregation) in every instantiation of the diagram. This property suggests the possible omission of an explicit generalization. Alternatively, if all instances of the more specific class are not supposed to be instances of the more general class, then something is wrong in the rest of the diagram, since it is forcing an undesired conclusion.
- Detection of redundancies, for example, if two classes (associations/aggregations) subsume each other, then one of them is redundant.
- Refinement of properties, for example, when the properties of various classes and associations/aggregations interact to yield stricter multiplicities or typing than those explicitly specified by the designer. The simplest cases arise with multiple inheritance.

The idea of using description logics to deal with resource models has its root in two considerations: (1) Both CIM models and resource models are expressed using UML class diagrams with specific conventions, and (2) DLR is able to fully capture UML class diagrams together with the specific conventions adopted in CIM. The approach can be summarized in the following way:

- The CIM core model and the parts of the common model relevant for a specific management task are expressed in terms of a DLR knowledge base.
- CIM-based resource models (actually, the dynamic model part) that are specific for components under analysis are in turn expressed as additional assertions included in the knowledge base.
- Queries posed to the CIM system specification are translated into queries to the corresponding DLR knowledge base, and are answered by making use of typical description logics reasoning tasks.

Representing CIM and resource models in terms of DLR allows for automated use of the knowledge represented in such models, making use of state-of-the-art reasoning tools developed for expressive description logics.

**Resource model knowledge exploitation.** The expressive power of description logics and their ability to capture and model a great variety of constraints allow for powerful reasoning operators to be applied across the CIM classes and CIM class properties representing a particular IT system. (Note that the standard UML representation of CIM schema does not allow this type of reasoning across classes.) We have successfully applied the capabilities offered by description logics in our work focused on the "automatic" definition of the dynamic model (aggregation of CIM classes) during the design phase. In our validation work we used two state-of-the-art description logic reasoning systems: FaCT[17,18] and RACER;[19] we formalized a CIM core model knowledge base given as input to FaCT and RACER in order to apply reasoning and inference operators. This technology appears suited to provide the reasoning engine for the resource model ontology service.

Referring again to the critical memory leak example (see Figure 2), CIM classes, attributes, and associations/aggregations are used in the resource model (actually, the dynamic model) to establish relationships between the various system components. Such relationships implicitly determine those that may contribute to the definition of a problem (e.g., "critical memory leak" between Memory and Process classes). If this model is loaded on the ontology server, the description logic reasoning engine can exploit it to determine relevant properties to be included in other resource models.

As an example, consider a new application installed on the managed endpoint with the requirement that resource leaks should be kept under control. This new application can be viewed as being composed of multiple processes. The monitoring engine asks the reasoning engine (by invoking suitable description logic inference requests) to identify the system components that may be involved in a resource leak. The reasoning system determines those classes that are subsumed by the description logic term representing the resource leak. In our example this is the

"critical memory leak" condition via the Process class. Thus, the model for the application includes the classes Memory and Process and the resource model class representing "critical memory leak." Other kinds of resource leaks could be similarly identified, such as "file descriptor leak" or "queue handle leak."

Further research into System Management Ontology aims at extending the description logic formalization from the domain of CIM classes and properties (dynamic model) to the domain of "best practices" and baselines (reference model). This may require extending the expressiveness of the adopted description logics (to include, for example, concrete domains) or taking into account the representation of extensional aspects and/or more expressive querying mechanisms.

The implications of this ongoing research effort are fascinating and challenging: we could apply reasoning mechanisms not only across classes and class properties but also across resource models. The reasoning mechanisms working across resource models could gracefully extend from the autonomic management of a single element (such as an operating system, a database, a business application) toward the autonomic management of an entire business solution.

In fact, because the resource model is fully represented via description logics, the ontology service can be used for reasoning across resource models, for detecting and correcting problems, for optimizing system behavior, or for achieving a specified quality-of-service (QoS). The derived resource model, or set of resource models, would then be "contextually pulled" to the relevant resource.

The use of description logics to represent and perform reasoning across resource models significantly augments the autonomic features of the IBM Tivoli Monitoring solution during the design and deployment phases. Figure 3 illustrates the basic flow when a managed resource uses SMO services.

Every time the managed resource undergoes a significant state change (such as when a new application is installed, or a new QoS is enforced on a running application) the monitoring agent uses the ontology service for reasoning about the appropriate management model for enforcing the desired QoS. This phase is controlled by means of user-defined policies that define the configuration for the monitoring agent, for example, "call the ontology service whenever application X is installed or when QoS is modified" (see items 1 through 3 in Figure 3).

The end result of the reasoning process performed by the ontology engine is the resource model (or set of resource models) best suited to address the specific condition at the target (e.g., "application X with QoS Q has been installed"). Default configuration parameters (cycle time, thresholds, etc.) are also applied to complete the definition of the reference model[20] (see item 4 in Figure 3).

The target knows (usually a URL is provided) where the resource model is located, or where it will be built. The resource model is then pulled to the target, typically in a Web environment (item 5 in Figure 3), where it exercises its problem-detecting capabilities (item 6 in Figure 3).

The described contextual pulling approach, with the support of an ontology service, represents a step toward the autonomic computing vision:

- The resource model can be selected and built, without human intervention, based on the context (state changes, specified QoS, etc.).
- The resource model is pulled "on demand" according to the state of the target.

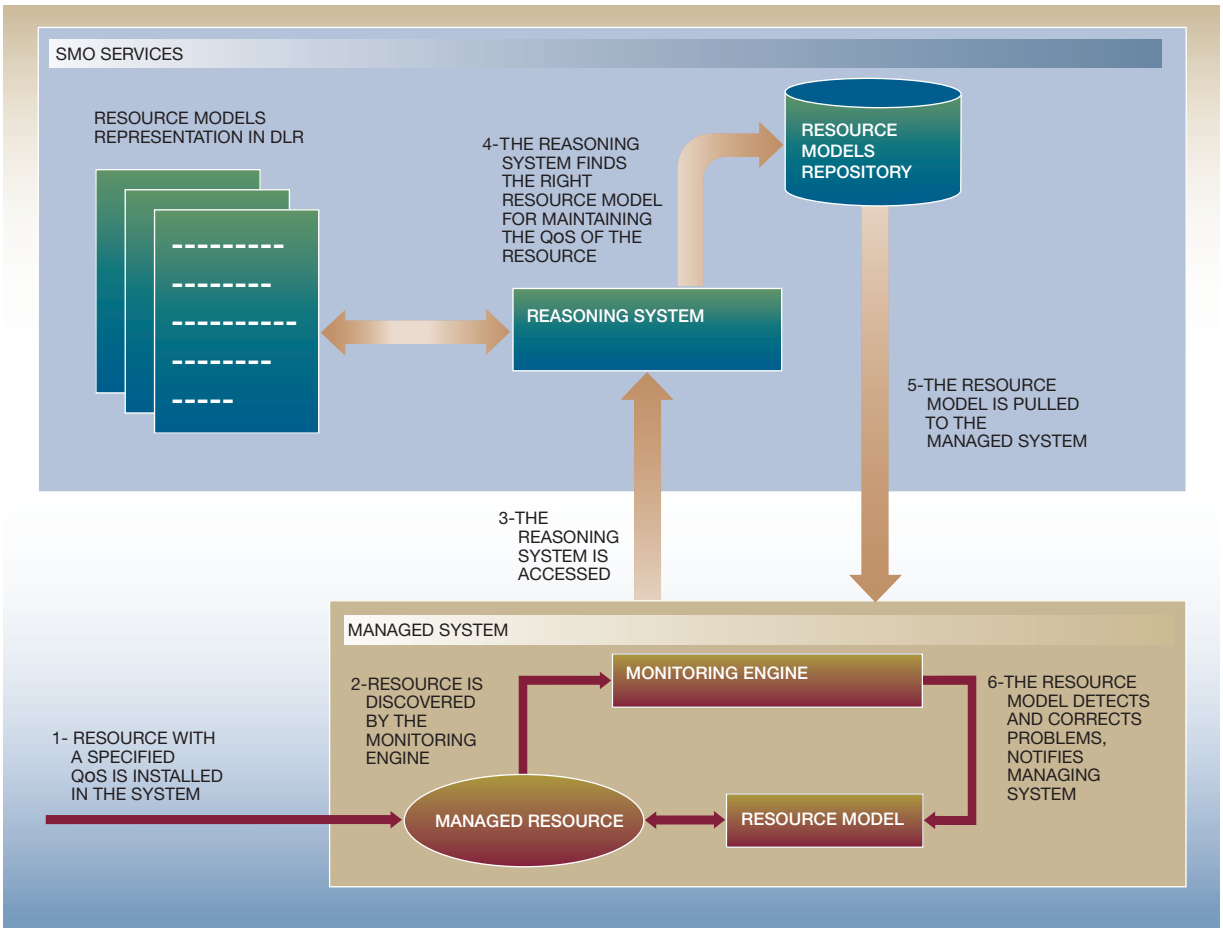
These enhanced autonomic capabilities do not intend to exclude completely the human intervention from the monitoring process. The human administrator will continue to be involved in the definition of the appropriate policies, in the supervision of the entire process, and in the tuning/correcting of specific "pathological" conditions.

## Conclusions

The resource model technology, currently implemented in the ITM product, helps realize the autonomic computing vision in systems management and specifically in the monitoring space: the resource model captures the key characteristics of a managed resource and allows the system to detect and correct any behavior not in line with the expected QoS.

In this paper, we proposed to use the resource model technology in order to extend the autonomic behavior across the entire life cycle of the application, including the design phase and the deployment phase. The CIM ontology and its representation by description logics raises the possibility to make better use

Figure 3 A managed resource using SMO services—the basic flow




of the knowledge available in CIM itself, and ultimately to perform automated reasoning on this knowledge base system. Finally, the contextual pulling approach is seen as an autonomic evolution of the management paradigm offered today by the Tivoli Management Environment:[5] it is the managed resource that, through the use of the SMO service, will be able to find and deploy the right resource model for keeping its QoS under control.

## Acknowledgments

The authors wish to thank Giuseppe De Giacomo, Richard Szulewski, Scot MacLellan, Daniela Berardi, Andrea Cali, and Maurizio Lenzerini for sharing their insight and for their suggestions on the topic of this paper.



## Cited references and notes

1. IBM Tivoli Monitoring, IBM Corporation, http://www.tivoli.com/products/index/monitor/.
2. Autonomic Computing, IBM Research Division, IBM Corporation, http://www.research.ibm.com/autonomic/.
3. The Distributed Management Task Force, Inc., http://www.dmtf.org/index.php.
4. Tivoli Management Framework 3.7.1, IBM Corporation, http://www.tivoli.com/support/public/Prodman/public_manuals/td/ManagementFramework3.7.1.html.
5. Tivoli Management Environment provides centralized management and a single point of control for data distribution to groups of systems. In the Tivoli environment, a *profile* provides a container for application-specific information about a particular type of resource. Profiles can be specialized and configured, distributed across a network, and applied to a di-

verse set of machines, according to a subscription paradigm. The profile feature is scalable with the number of distributed systems.
6. D. Fensel, *Ontologies: A Silver Bullet for Knowledge Management and Electronic Commerce*, Springer, New York (2001).
7. D. Fensel, F. van Harmelen, I. Horrocks, D. L. McGuinness, and P. F. Patel-Schneider, "OIL: An Ontology Infrastructure for the Semantic Web," *IEEE Intelligent Systems* **16**, No. 2, 38–45 (2001).
8. D. Calvanese, M. Lenzerini, and D. Nardi, "Unifying Class-Based Representation Formalisms," *Journal of Artificial Intelligence Research* **11**, 199–240 (1999).
9. *The Description Logic Handbook: Theory, Implementation and Applications*, F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. F. Patel-Schneider, Editors, Cambridge University Press (2002). To appear.
10. D. Calvanese, G. De Giacomo, and M. Lenzerini, "On the Decidability of Query Containment under Constraints," *Proceedings of the 17th ACM SIGACT SIGMOD SIGART Symposium on Principles of Database Systems* (PODS'98), ACM, New York (1998), pp. 149–158.
11. D. Calvanese, G. De Giacomo, M. Lenzerini, D. Nardi, and R. Rosati, "Description Logic Framework for Information Integration," *Proceedings of the 6th International Conference on Principles of Knowledge Representation and Reasoning* (KR'98), Morgan Kaufmann Publishers, San Francisco, CA (1998), pp. 2–13.
12. D. Calvanese, G. De Giacomo, and M. Lenzerini, "Identification Constraints and Functional Dependencies in Description Logics," *Proceedings of the 17th International Joint Conference on Artificial Intelligence* (IJCAI 2001), Morgan Kaufmann Publishers, San Francisco, CA (2001), pp. 155–160.
13. J. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Modeling Language Reference Manual*, Addison-Wesley Publishing Co., Boston, MA (1998).
14. A. Calì, D. Calvanese, G. De Giacomo, and M. Lenzerini, "A Formal Framework for Reasoning on UML Class Diagrams," *Proceedings of the 13th International Symposium on Methodologies for Intelligent Systems* (ISMIS 2002), Lecture Notes in Computer Science 2366, Springer, New York (2002), pp. 503–513.
15. D. Berardi, D. Calvanese, and G. De Giacomo, "Reasoning on UML Class Diagrams Using Description Logic Based Systems," *Proceedings of the KI'2001 Workshop on Applications of Description Logics*, CEUR Electronic Workshop Proceedings (2001), http://ceur-ws.org/Vol-44/.
16. A. Calì, D. Calvanese, G. De Giacomo, and M. Lenzerini, "Reasoning on UML Class Diagrams in Description Logics," *Proceedings of IJCAR Workshop on Precise Modelling and Deduction for Object-Oriented Software Development* (PMD'01), (2001), http://i12www.ira.uka.de/~beckert/PMD/proceedings.html.
17. I. Horrocks, "The FaCT System," H. C. M. de Swart, Editor, *Proceedings of the 2nd International Conference on Analytic Tableaux and Related Methods* (TABLEAUX'98), Lecture Notes in Artificial Intelligence 1397, Springer, New York (1998), pp. 307–312.
18. I. Horrocks, "Using an Expressive Description Logic: FaCT or Fiction?" *Proceedings of the 6th International Conference on Principles of Knowledge Representation and Reasoning* (KR'98), Morgan Kaufmann Publishers, San Francisco, CA (1998), pp. 636–649.
19. V. Haarslev and R. Moeller, "RACER System Description," *Proceedings of the International Joint Conference on Automated Reasoning* (IJCAR 2001), Lecture Notes in Artificial Intelligence 2083, Springer, New York (2001), pp. 701–705, http://www.globus.org/OGSA
20. The most appropriate values for the configuration parameters heavily depend on the operational conditions at the target. In ITM Version 5.1 a default setting is given, which the user may change during the configuration phase. In the future, we envision an Adaptive Manager running at the target whose function is to tune the configuration parameters of the resource models according to the operational conditions of the hosting environment.

*Accepted for publication September 20, 2002.*

**Giovanni Lanfranchi** *IBM Software Group, Rome Tivoli Laboratory, Via Sciangai 53, I-00144 Roma, Italy (electronic mail: Giovanni_Lanfranchi@it.ibm.com).* Mr. Lanfranchi is currently technical strategist for the Performance and Availability department in Tivoli. He received his degree in physics in 1986 at the University of Milan. Before joining IBM he was responsible for research and development at several companies in the robotics and automotive field. At IBM he was an architect in the Computer-Aided Engineering department, where he developed leading-edge technologies in the surface modeling area. Mr. Lanfranchi is currently with the IBM Tivoli Division where he is involved in the design of the "new wave" of systems management applications with specific focus on performance and availability.

**Pietro Della Peruta** *IBM Software Group, Rome Tivoli Laboratory, Via Sciangai 53, I-00144 Roma, Italy (electronic mail: pietro_della_peruta@it.ibm.com).* Mr. Della Peruta is currently principal engineer in the performance and availability unit of IBM Tivoli. In this role he is responsible for technical strategy and implementation of performance and availability management applications. Previously he worked on the design of the IBM Tivoli Monitoring product. He joined Tivoli in 1996 after six years as technical architect of IBM performance products such as NetView Performance Monitor. He holds a B.A. degree in electronics engineering from the University of Naples.

**Antonio Perrone** *IBM Software Group, Rome Tivoli Laboratory, Via Sciangai 53, I-00144 Roma, Italy (electronic mail: antonio.perrone@it.ibm.com).* Mr. Perrone graduated *cum laude* in information sciences in 1989 at the University of Bari, Italy. He joined IBM in 1990, where he has held several positions in development and support. He has been with the IBM Tivoli Division since 1996, where he is currently an architect of IBM Tivoli Monitoring. His interests include systems management, performance, and availability, knowledge representation, data integration, transaction and process modeling, and software development.

**Diego Calvanese** *Università di Roma "La Sapienza," Dipartimento di Informatica e Sistemistica, Via Salaria 113, I-00198 Roma, Italy (electronic mail: calvanese@dis.uniroma1.it).* Dr. Calvanese is Assistant Professor in the Department of Informatics and Systems of the University of Rome, where he received his Ph.D. in computer science in 1995. His research interests include theoretical aspects of data and information integration, semi-structured data, logics for knowledge representation, and in particular description logics and their relationship to data models.

# Competitive algorithms for the dynamic selection of component implementations

by D. M. Yellin

As component-based development matures, more and more applications are built by integrating multiple distributed components. We suggest providing components with multiple implementations, each optimized for a particular workload, and augmenting the component run-time environment with a mechanism for switching between implementations. This mechanism monitors the types of requests the component is receiving, and adaptively switches implementations for optimal application performance. Achieving this optimal performance depends on making good choices as to when and how to switch implementations, a problem we refer to as the *adaptive component problem*. We first formalize the generic problem and then provide an algorithm, named Delta, for switching implementations in the special case when the component has exactly two implementations. We show that this algorithm is (3 + epsilon)-competitive with respect to the optimal algorithm, where epsilon is a small fraction. We establish a 3-competitive lower bound for the problem, which implies that Delta is close to optimal. We describe the application of these results to the distributed pub/sub problem, and the data structure selection problem.

Many applications are being built by integrating multiple distributed components in order to implement a particular business function. The increasing popularity of component programming models, such as JavaBeans**[1] and Web services,[2] is predicted to further accelerate the adoption of *component-based development* (CBD).

An impediment to the adoption of CBD, however, is the inability of the "user" of the components to optimize their performance for use in a particular solution. Web services, with its premise of loosely coupled distributed components working together, makes this issue even more acute, as the development, deployment, and maintenance of components making up a single solution may come from different vendors and run in very different system environments.

In this paper, we propose a generic framework that addresses this issue. In our formulation, an *adaptive component* has multiple implementations, each optimized for a particular request workload. A mechanism for switching between implementations is also provided. The underlying system monitors the current request workload for a given component and adaptively switches to the implementation best suited to this workload. By shifting more of the performance optimization burden to the component, performance tuning is simplified and the resulting overall system performance is likely to be improved. Additionally, by having the system monitor the request workload and dynamically switch implementations based upon

current usage patterns, the component can better accommodate changing request patterns.

Because switching between implementations can incur a heavy cost, good algorithms are needed for determining, at run time, when to switch between implementations. We call this the *adaptive component problem*. This paper describes an algorithm, named Delta, for the case when there are exactly two implementations. We show that Delta is $(3 + \epsilon)$-competitive. This algorithm is designed for the case when the cost of switching between implementations is very large compared to the cost of processing a single request. In this case, the value of $\epsilon$ is guaranteed to be very small. Because we also show a 3-competitive lower bound, this algorithm is close to optimal.

We show the applicability of this framework to two problems. The first is an adaptive version of the distributed *pub/sub* problem, where multiple loosely coupled components are reading and writing from a shared data repository. A component can either read and write to this shared repository or create a local data cache for fast access. The second example is an adaptive version of the *data structure selection* problem, where an application must choose the appropriate internal data structure to use in order to provide the quickest answer to particular queries. We show that both of these are instances of the adaptive component problem and that Delta can be used to decide when to switch implementations, thus optimizing run-time performance. This illustrates the applicability of our framework to a wide range of problems.

Here is an outline for the rest of this paper. The next section "The adaptive component problem" is followed by "The Delta algorithm." Then, the section "Examples" contains our two case studies. In the section "Competitiveness" we prove that Delta is $(3 + \epsilon)$-competitive, and in the section "Lower bounds," we establish a 3-competitive lower bound for this problem. The section "Related work" is followed by "Summary and open issues."

## The adaptive component problem

We propose a model where the component developer implements multiple versions of a component, each optimized for a specific request workload. The developer also writes the code that controls the switching between implementations at run time. For each request type that the component can service, and for each implementation of that component, we determine the cost for processing a request of that type. We similarly determine the cost of switching between any two implementations. These costs may be specified by the component developer or may be derived empirically (e.g., by profiling). An on-line algorithm will monitor request workloads at run time and determine when to switch implementations. The rest of this section formalizes these concepts and defines an optimality criterion for determining when to switch implementations.

Let *Comp* be an adaptive component, and let $ReqTypes = \{typ_i\}$ be the set of request types that *Comp* can process. Let $Impls = \{impl_j\}$ be the set of implementations of *Comp*, with $impl_1$ being the *default* implementation. Let $Cost : ReqTypes \times Impls \rightarrow \mathbf{R}$ be the function that gives the cost for *Comp* to process a request of a given type using a particular implementation ($\mathbf{R}$ denotes the set of reals). Let $SwitchCost : Impls \times Impls \rightarrow \mathbf{R}$ be the function that determines the cost in *Comp* of switching from one implementation to another. $SwitchCost(impl_i, impl_j) = \infty$ iff *Comp* cannot switch from $impl_i$ to $impl_j$. The cost functions may reflect internal computation costs, network message costs, or a combination of these and other metrics, depending upon the application environment.

We represent the computation to be performed as a sequence of requests, $r_1, \cdots, r_k$. The empty sequence is denoted by $\Omega$. To facilitate the modeling of many different sorts of problems, we allow a single request to be processed by either a single component or by multiple components. Given such a sequence of requests, a switch of implementations may occur after processing any request in the sequence. We represent this by the operation $switch(impl_i, impl_j)$, where $impl_i$ is the current implementation of the component, and $impl_j$ is the implementation being switched to. Hence, given a sequence of requests $\alpha = r_1, \cdots, r_k$, we model the adaptive behavior of the component *Comp* as transforming this sequence to the sequence $\alpha' = s_1, \cdots, s_f$ such that the following comments are true:

- For $1 \leq i \leq f$, either $s_i$ is a request or $s_i$ is a switch operation.
- Removing the switch operations from $\alpha'$ produces $\alpha$.
- If $s_m = switch(impl_j, impl_k)$ then either $s_m$ is the first switch operation in the sequence and $j = 1$ or the closest preceding switch operation in the sequence is of the form $switch(impl_i, impl_j)$.

An *on-line* algorithm is one that transforms the sequence $\alpha$ by deciding whether or not to insert a switch operation after request $r_i$ based only upon the sequence seen so far, $r_1, \cdots, r_i$. We write $\alpha \rightarrow_A \alpha'$ to indicate that on-line algorithm $A$ (used by *Comp*) maps $\alpha$ into $\alpha'$ when processing it. For any request $s_i$ in the transformed sequence $\alpha'$, we say that the *implementation* $impl_k$ *is active at* $s_i$ if the closest *switch* operation preceding $s_i$ in the sequence is of the form $switch(impl_j, impl_k)$, or $k = 1$ and there is no *switch* operation preceding $s_i$. When algorithm $A$ and sequence $\alpha$ are understood from the context, we simply denote this by $ImplIs(s_i, impl_k)$.

$Cost^A_\alpha$ is the cost of *Comp* to process request sequence $\alpha$ using algorithm $A$. $Cost^A_\alpha = Cost(\alpha') = \sum_{i=1}^{n} C(s_i)$ where $\alpha \rightarrow_A \alpha' = s_1, \cdots, s_n$ and where

$$C(s_i) = \begin{cases} Cost(s_i, impl_k) \\ \quad \text{if } s_i \text{ is a request, and } implIs(s_i, impl_k); \\ SwitchCost(impl_j, impl_k) \\ \quad \text{if } s_i \text{ is the operation } switch(impl_j, impl_k). \end{cases}$$

Let $O$ be an optimal algorithm; that is, for any sequence $\alpha$ and any on-line algorithm $A$, $Cost^O_\alpha \leq Cost^A_\alpha$. An on-line algorithm $A$ is *c-competitive*[3,4] iff, for any sequence $\alpha$, there exist constants $c$ and $d$ such that $Cost^A_\alpha \leq c * Cost^O_\alpha + d$. Given *Comp, RegTypes, Impls*, *Cost*, and *SwitchCost*, the *adaptive component problem* is to find an on-line competitive algorithm for this problem instance, that is, to find a *c-competitive* algorithm for some constant $c$.

Now consider a special case of the adaptive component problem where $Impls = \{impl_1, impl_2\}$, which we call the adaptive *two-implementation-component problem* (the adjective "adaptive" is sometimes omitted, for conciseness). We assume that there is at least one request $r$ such that $Cost(r, impl_1) < Cost(r, impl_2)$, and at least one request $r'$ such that $Cost(r', impl_1) > Cost(r', impl_2)$. Also, we assume that there are no constraints on the order in which requests must be composed in a sequence (i.e., any interleaving of requests is a legitimate request sequence).

Let $SC_1 = SwitchCost(impl_1, impl_2)$, $SC_2 = SwitchCost(impl_2, impl_1)$, and $SC = SC_1 + SC_2$. We call $SC$ the *round trip switching cost*. The Delta algorithm described in the next section will perform close to optimal when, for any request $r$, $|Cost(r, impl_1) - Cost(r, impl_2)| \ll SC$. That is, the difference in processing cost for a single request when one implementation is active as opposed to the other implementation, is significantly less than the switching cost.

## The Delta algorithm

Consider any two-implementation-component problem, where $SC$ is the previously defined round trip switching cost. Let $\alpha = r_1, \cdots, r_k$ be a sequence of requests. Then $Cost(\alpha, impl_j) = \sum_{i=1}^{k} Cost(r_i, impl_j)$, $j = 1, 2$. Algorithm Delta is extremely simple and works as follows. Say that implementation $impl_i$ of component *Comp* is active, $i \in \{1, 2\}$, and the processing of request $r_k$ in the sequence $r_1, r_2, \cdots$ has just ended. Given $impl_i$, denote the other implementation by $\overline{impl_i}$. If there exists $j \leq k$ such that $Cost((r_j, \cdots, r_k), \overline{impl_i}) \leq Cost((r_j, \cdots, r_k), impl_i) - SC$, then *Delta* instructs *Comp* to switch implementations.

Delta also has a simple implementation using just three counters: *Impl*1*Cost*, *Impl*2*Cost*, and *MinDelta*. The algorithm, when $impl_1$ is assumed active, is given in Figure 1. The algorithm works analogously when $impl_2$ is active.

To see why this implementation is correct, note that by definition $Cost(r_0) = 0$ and if Delta switches to $impl_2$ after $r_k$, then $\exists j$, $1 \leq j \leq k$, such that

$$\begin{aligned} &(Cost((r_1, \cdots, r_k), impl_1) \\ &\qquad - Cost((r_0, \cdots, r_{j-1}), impl_1)) \\ &-(Cost((r_1, \cdots, r_k), impl_2) \\ &\qquad - Cost((r_0, \cdots, r_{j-1}), impl_2)) \geq SC \\ &\Leftrightarrow \\ &(Cost((r_1, \cdots, r_k), impl_1) \\ &\qquad - Cost((r_1, \cdots, r_k), impl_2)) \\ &-(Cost((r_0, \cdots, r_{j-1}), impl_1) \\ &\qquad - Cost((r_0, \cdots, r_{j-1}), impl_2)) \geq SC \end{aligned}$$

After processing request $k$ in the sequence, the value of *MinDelta* in the implementation of Figure 1 equals the minimum of 0 and $\min_{0 \leq j \leq k} \{Cost((r_0, \cdots, r_j), impl_1) - Cost((r_0, \cdots, r_j), impl_2)\}$.

At this same point *Impl*1*Cost* equals $Cost((r_1, \cdots, r_k), impl_1)$, and *Impl*2*Cost* equals $Cost((r_1, \cdots, r_k), impl_2)$. Hence the equation above is true iff:

$$(Impl1Cost - Impl2Cost) - MinDelta \geq SC$$

This is exactly the computation that the algorithm of Figure 1 performs in order to determine when to switch implementations.

Figure 1 The Delta Algorithm ($impl_1$ assumed active)

```
Impl1Cost := 0;  Impl2Cost := 0; MinDelta := 0;
TimeToSwitch := false;
While (not TimeToSwitch) {
        Process next request r;
        Impl1Cost := Impl1Cost + Cost(r,impl1);
        Impl2Cost := Impl2Cost + Cost(r,impl2);
        Temp := Impl1Cost – Impl2Cost;
        MinDelta := min(MinDelta,Temp);
        If ((Impl1Cost – Impl2Cost – MinDelta) > = SC)
          {TimeToSwitch := true;}
}
Switch to impl2;
```

The section "Competitiveness," later, contains a formal proof that Delta is competitive and close to optimal. We now provide some intuition as to why this is the case. Let $A$ be any algorithm and consider $B$, the "adversary" of $A$, an algorithm that observes $A$ and tries to devise a sequence on which $A$ performs worse than $B$. If $A$ keeps $impl_1$ active too long, the adversary will devise the sequence to be costly when $impl_1$ is active and cheap when $impl_2$ is active. Hence, any competitive algorithm must keep switching between implementations when it determines that the cost of keeping the other implementation active is lower. But if $A$ switches implementations too often, the switching costs will dominate and the adversary can choose to keep the same implementation active, thus avoiding all switching costs. The key insight of Delta is to switch exactly when it accumulates $SC$ more in cost than the other implementation would, if it were active. Choosing any value other than $SC$ results in worse performance.

To see why this is the case, say that $A$ makes $impl_1$ active and switches to $impl_2$ after accumulating additional cost $k$, $k < SC$ (more than it would have accumulated if $impl_2$ were active). As soon as $A$ makes $impl_2$ active, the adversary would design the sequence so that $A$ accumulates additional cost $k$ when $impl_2$ is active. At that point $A$ will switch back to $impl_1$. The adversary would keep $impl_1$ active throughout this sequence. Hence, the cost to the adversary would be just $k$, while $A$'s cost would be $k + SC + k > 3k$. $A$ would thus be worse than 3-*competitive* (how much worse would depend upon the choice of $k$).

On the other hand, say that $A$ makes $impl_1$ active and $A$ switches to $impl_2$ after accumulating additional cost $k$, $k > SC$. The adversary would make $impl_2$ active at the start of the sequence. As soon as $A$ accumulates $k$ in cost and switches to $impl_2$, the adversary would switch to $impl_1$ and would design the sequence so that $A$ accumulates additional cost $k$ when $impl_2$ is active. At that point $A$ will switch back to $impl_1$. Hence, the cost to the adversary would be just $SC$, while $A$'s cost would be $k + SC + k > 3 * SC$. Hence, $A$ would be worse than 3-*competitive*.

The reason Delta is $(3 + \epsilon)$-competitive and not just 3-competitive is due to *boundary* conditions; Delta does not always have the opportunity to switch when it accumulates exactly $SC$ more in cost. For instance, in may be that $impl_1$ is active and Delta has accumulated $SC - 1$ more in cost than it would have if $impl_2$ had been active. The next request $r$ may cost it $rCost = max_r Cost(r, impl_1) - Cost(r, impl_2)$ more to process in $impl_1$ than in $impl_2$. Therefore Delta will not actually switch until it accumulates $SC + rCost - 1$ more cost in $impl_1$ than it would have in $impl_2$. The $\epsilon$ term bounds the cost of these boundary conditions.

## Examples

In this section we apply the Delta algorithm to two problems: the distributed pub/sub problem and the data structure selection problem.

**The distributed pub/sub problem.** Consider a data server that serves records from a database to many clients. The server and each client reside on separate nodes of the network. Each client can perform a read or a write on the data. We assume that the clients are independent, that is, a client reads and writes data independent of any other client. There is no synchronization between clients. More formally, for any given "run," there is a linear order in which each client reads and writes records, but the interleaving at the server of reads and writes from different clients is subject to various factors, and cannot be predicted.

Each client can exist in one of two modes: either in *subscription* (Sub) mode or in *nonsubscription* (Non-Sub) mode. In the latter case, for each read that the client wants to perform, it must send a message to the server and receive a reply back. In Sub mode, a client caches a local copy of the database. All reads of the database go against this local copy. In either

case, writes must still go to the server. Upon receiving a write update from any client, the server must inform all subscribers (even the writer of the data if he or she is in Sub mode) of the change to the data. This mechanism is known as "pub/sub" (publication/subscription).

Because it is often impossible to statically predict the read/write behavior of clients, and because their behavior changes over time, we consider an adaptive pub/sub strategy that does not require a client to permanently use either Sub or NonSub mode, but that can flexibly switch between these implementations depending upon current workloads. The goal is to find an optimal strategy for switching implementations that minimizes network traffic.

An instance of the pub/sub problem consists of a server and $m$ clients denoted by $client_j$ $(1 \leq j \leq m)$. Let $r_1, \cdots, r_k$ be a sequence of requests where request $r_j$ is either $read_j$ or $write_j$, $1 \leq j \leq m$, indicating that $client_j$ is either reading or writing a data record to the database. We assume that the database contains $p - 1$ records (by database we simply mean some collection of data items, where each item can be read and written individually). We now focus on how the requests related to $client_i$, $1 \leq i \leq m$, when in NonSub mode, are carried out.

- $read_i(r)$: This request generates a message from $client_i$ to the server asking for record $r$, and a message back from the server delivering record $r$.
- $read_j(r), j \neq i$: This request does not concern $client_i$ (it is only relevant to $client_j$).
- $write_i(r)$: This request generates a message from $client_i$ to the server asking for record $r$ to be written.
- $write_j(r), j \neq i$: This request does not concern $client_i$.

The requests related to $client_i$, $1 \leq i \leq m$, when in Sub mode are carried out as follows.

- $read_i(r)$: Record $r$ is read from the cached data at $client_i$, thereby avoiding the need for messages to the server.
- $read_j(r), j \neq i$: This request does not concern $client_i$.
- $write_i(r)$: This request generates a message from $client_i$ to the server asking for record $r$ to be written, and an $update(r)$ message from the server to $client_i$.

Figure 2 Cost to $client_i$ of processing requests $(j \neq i)$

$$
\begin{aligned}
Cost(write_i, NonSub) &= 1\\
Cost(write_j, NonSub) &= 0\\
Cost(read_i, NonSub) &= 2\\
Cost(read_j, NonSub) &= 0\\
SwitchCost(NonSub, Sub) &= p\\
\\
Cost(write_i, Sub) &= 2\\
Cost(write_j, Sub) &= 1\\
Cost(read_i, Sub) &= 0\\
Cost(read_j, Sub) &= 0\\
SwitchCost(Sub, NonSub) &= 1
\end{aligned}
$$

- $write_j(r), j \neq i$: This request generates an $update(r)$ message from the server to $client_i$ informing it that record $r$ has been updated.

Switching implementations is carried out through subscribe and unsubscribe messages.

- *subscribe*( ): This is shorthand for *switch*(*NonSub, Sub*). It generates a message from $client_i$ to the server, subscribing $client_i$ to the database, and a message from the server delivering a local copy of the database to $client_i$. It puts $client_i$ into Sub mode.
- *unsubscribe*( ): This is shorthand for *switch*(*Sub*, *NonSub*). It generates a message from $client_i$ to the server unsubscribing $client_i$. It puts $client_i$ into NonSub mode, and discards its local copy of the database.

The cost of a request is proportional to the number of messages it generates and their size, such that sending a single message containing a single record has unit cost. Figure 2 gives the costs to $client_i$ for each of these requests.

Algorithm Delta can be applied to the pub/sub problem by monitoring each read and write request at $client_i$, switching from NonSub to Sub mode when it detects that Sub mode would have processed a previous subsequence of the requests for $p + 1$ less in network messaging costs than it was processed in NonSub mode. The number $p + 1$ is the switch cost of the Delta algorithm; $SwitchCost(NonSub, Sub) + SwitchCost(Sub, NonSub) = p + 1$. It monitors the requests being processed when in Sub mode and de-

Figure 3 A read/write sequence (α) and a transformed sequence (α')

$\alpha = read_1, write_1, read_1, read_2, read_1, read_1, read_1, write_2, write_1, read_1, read_2, write_2, read_1, read_1, write_2, read_1, read_1$

$\alpha' = subscribe_1, localRead_1, write_1, update_1, localRead_1, read_2, localRead_1, localRead_1, localRead_1, write_2, update_1, write_1, update_1, localRead_1, read_2, write_2, update_1, localRead_1, localRead_1, write_2, update_1, localRead_1, localRead_1$

Figure 4 The processing of sequence α by algorithm Delta

$\alpha = read_1, write_2, write_2, write_2, write_2, read_1, read_1, read_1, write_2, read_1$

| Op Number x | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| *Impl1Cost* | 0 | 2 | 2 | 2 | 2 | 2 | 4 | 6 | 8 | 8 | 10 |
| *Impl2Cost* | 0 | 0 | 1 | 2 | 3 | 4 | 4 | 4 | 4 | 5 | 5 |
| *Impl1Cost – Impl2Cost* | 0 | 2 | 1 | 0 | –1 | –2 | 0 | 2 | 4 | 3 | 5 |
| *MinDelta* | 0 | 0 | 0 | 0 | –1 | –2 | –2 | –2 | –2 | –2 | –2 |
| (*Impl1Cost – Impl2Cost*) – *MinDelta* | 0 | 2 | 1 | 0 | 0 | 0 | 2 | 4 | 6 | 5 | 7 |

cides when to switch to NonSub mode based upon a similar calculation.

Figure 3 gives a sequence $\alpha$ involving two clients and a server. For $client_1$, $Cost(\alpha, NonSub) = 22$ and $Cost(\alpha, Sub) = 7$. If $p + 1 \leq 15$, then Delta would dictate that $client_1$ should subscribe in the course of this sequence. $\alpha'$ is the transformed sequence, if $client_1$ were to subscribe at the beginning of this sequence. In this example, $op_i$ indicates that $client_i$ is performing operation $op$, where $op \in \{read, localRead, write, subscribe, unsubscribe\}$, $i \in \{1, 2\}$, and $update_i$ indicates that the server is sending an update message to $client_i$. *localRead* indicates a *read* operation from the local cache.

There is one subtlety, however, in using Delta for the pub/sub problem. Looking at Figure 2, we see that Delta needs to know not only the reads and writes done by $client_i$, but also the writes done by any $client_j$, $j \neq i$. If Delta is running at $client_i$, how does it get this information? One possibility is to have Delta run at the server, where all client reads and writes are known, and have the server tell $client_i$ when to switch implementations. However, it may be advantageous to have the control for switching implementations located at the client, thereby providing a more distributed architecture.

In Reference 5 we address the question "How can Delta, running at $client_i$, know about the write operations performed by other clients without increasing the number of messages that must be exchanged between the client and the server?" In Sub mode the issue does not exist, as the client can infer this from the number of updates messages it receives. In NonSub mode, it is required that the server piggyback a count of the total number of write operations it has received on the reply to each read request by $client_i$.

Figure 4 illustrates the processing by Delta of sequence $\alpha$ involving $client_1$, $client_2$, and a server. The value of the counters as well as the intermediate computations at $client_1$ (in NonSub mode) are shown, as calculated by Delta (cf. Figure 1). In Figure 4, $impl_1$ is in NonSub mode while $impl_2$ is in Sub mode.

**The adaptive data structure selection problem.** Often an application chooses a data structure that is optimized for particular requests. When different types of requests require different data structures, either duplicate data structures are maintained, or a single data structure is used, chosen presumably for the most frequent type of requests. Consider, for example, a set of records $S$ that has two possible keys, $i_1$ and $i_2$. (My bank allows me to identify myself either by my social security number or by my account number.) Let *Comp* be the component that encapsulates all operations on $S$. *Comp* may support many types of requests that operate on this set, but all of these requests can be categorized as either of type $r_1$, for requests using key $i_1$, or of type $r_2$, for requests using key $i_2$.

One strategy is for *Comp* two create two index tables for $S$, one using key $i_1$ and one using key $i_2$. Each index table is implemented as a dictionary,[6] allowing one to perform the usual operations of finding, inserting, and deleting elements. But this strategy may not always be feasible, because there may not be sufficient capacity for storing both indices (e.g., a pervasive device), or the overhead in maintaining these two separate indices may be too high. In such cases, we can support two implementations, one that uses key $i_1$, and one that uses key $i_2$. Algorithm Delta is used to dynamically decide when to switch between these implementations. In Reference 5 we present a more complete discussion on the use of Delta in this context.

## Competitiveness

For any two-implementation-component problem, let $SC_1 = SwitchCost(impl_1, impl_2)$, $SC_2 = SwitchCost(impl_2, impl_1)$, and let $SC$ be the round trip switching cost $SC = SC_1 + SC_2$. Given request $r$ let

$$reqCost = max_r|Cost(r, impl_1) - Cost(r, impl_2)|.$$

Let $\epsilon = 2{*}reqCost/SC$. In this section we prove the following theorem:

**Theorem 1:** *Algorithm Delta is* $(3 + \epsilon)$*-competitive for any two-implementation-component problem.*

Note that when the cost of processing a single request is significantly less than the cost of switching implementations, then $\epsilon \ll 1$ and Delta is close to optimal.

**Proof:** Consider any sequence $\alpha$ processed by Delta. $\alpha$ can always be viewed as consisting of recurring segments of the form $\omega = \beta, switch(impl_1, impl_2), \gamma, switch(impl_2, impl_1)$, where $\beta$ and $\gamma$ are sequences of requests, during processing of $\beta$ $impl_1$ is active, and during processing of $\gamma$ $impl_2$ is active. Note that $\beta$ and $\gamma$ contain no *switch* statements. Without loss of generality, we can assume that $\beta, \gamma \neq \Omega$ ($\Omega$ is the empty sequence). We will maintain the following invariant:

*Invariant:* At the start of processing of an $\omega$ segment, for *any* algorithm processing this sequence, either $impl_1$ is active or $impl_2$ is active and there is a *debt* of $SC_1$. By "debt" we mean that if, at the start of processing of a segment $\omega$ $impl_2$ is active, then a cost $SC_1$ has accrued in a preceding segment that has not yet been charged to the algorithm. Initially we know this is true because we assume that all algorithms start in $impl_1$ mode.

Let $r$ be the last request in $\beta$. Note that $Cost(r, impl_1) \leq Cost(r, impl_2) + reqCost$. There are two properties of note regarding the cost to any algorithm of processing the $\beta$ portion of segment $\omega$.

*Property 1:* Let $\mathcal{A}$ be any algorithm that processes $\beta$ with $impl_1$ active at both the start and the end of the sequence, and switching between implementations any number of times in between. Then $Cost_\beta^{\mathcal{A}} > Cost(\beta, impl_1) - reqCost$; that is, at most *reqCost* is gained by switching back and forth during the processing of $\beta$. To see why this is true, consider the case when $\mathcal{A}$ switches only once; say, $\mathcal{A}$ makes $impl_1$ active during $\beta_1$, $impl_2$ active during $\beta_2$, and $impl_1$ active during $\beta_3$, where $\beta = \beta_1, \beta_2, \beta_3$. Assume that $\beta_2 \neq \Omega$, otherwise this is trivially true. If $\beta_3 \neq \Omega$ then $Cost(\beta_2, impl_2) + SC > Cost(\beta_2, impl_1)$, otherwise Delta would have switched to $impl_2$ before or at the end of processing $\beta_2$. If $\beta_3 = \Omega$ then let $\beta_2'$ be the prefix of $\beta_2$ with one operation ($r$) removed from the end of $\beta_2$. Then $Cost(\beta_2', impl_2) + SC > Cost(\beta_2', impl_1)$ since Delta did not switch implementations before completing the processing of $\beta$. Since $Cost(r, impl_2) \geq Cost(r, impl_1) - reqCost$, we have

$$\begin{aligned} &Cost(\beta_2, impl_2) + SC \\ &\quad = Cost(\beta_2', impl_2) + Cost(r, impl_2) + SC \\ &\quad > Cost(\beta_2', impl_1) + Cost(r, impl_1) - reqCost \\ &\quad = Cost(\beta_2, impl_1) - reqCost \end{aligned}$$

Hence,

$$Cost_{\beta}^{\mathcal{A}} = Cost(\beta_1, impl_1) + Cost(\beta_2, impl_2) + Cost(\beta_3, impl_1) + SC > Cost(\beta, impl_1) - reqCost$$

A similar argument applies to any algorithm that switches multiple times back and forth between implementations during $\beta$. This completes the proof of Property 1.

*Property 2:* For any $\beta_1$, $\beta_2$ such that $\beta = \beta_1\beta_2$,

$$Cost(\beta, impl_1) < Cost(\beta_1, impl_1) + Cost(\beta_2, impl_2) + SC + reqCost$$

That is, it costs at most $SC + reqCost$ more to process $\beta$ when $impl_1$ is active than processing part of it when $impl_1$ is active and the rest when $impl_2$ is active. If $\beta_2 = \Omega$, this is trivially true. So assume that $\beta_2 \neq \Omega$ and let $\beta_2'$ be the prefix of $\beta_2$ with one operation ($r$) removed from the end of $\beta_2$. Then

$$Cost(\beta_1\beta_2', impl_1) < Cost(\beta_1, impl_1) + Cost(\beta_2', impl_2) + SC$$

since Delta did not switch implementations before completing the processing of $\beta$. Since

$$Cost(r, impl_1) \leq Cost(r, impl_2) + reqCost,$$

we have

$$\begin{aligned} &Cost(\beta, impl_1) \\ &\quad = Cost(\beta_1\beta_2', impl_1) + Cost(r, impl_1) \\ &\quad < Cost(\beta_1, impl_1) + Cost(\beta_2', impl_2) + SC \\ &\qquad + Cost(r, impl_2) + reqCost \\ &\quad = Cost(\beta_1, impl_1) + Cost(\beta_2, impl_2) + SC \\ &\qquad + reqCost \end{aligned}$$

This completes the proof of Property 2.

Let $l_1 = Cost(\beta, impl_1) - SC - reqCost$. Now we determine the cost to any adversary algorithm *Adv* of processing $\beta$ by examining the four cases A through D.

A: *Adv* starts with $impl_1$ active and ends with $impl_1$ active. If it never switches implementations, its cost is $l_1 + SC + reqCost$. Otherwise, say that it switches from $impl_1$ to $impl_2$ and back to $impl_1$ during $\beta$. By Property 1, this can only decrease its cost by *reqCost*, so its cost is at least $l_1 + SC$. Switching multiple times will not further decrease its cost.

B: *Adv* starts with $impl_1$ active and ends with $impl_2$ active. By Property 2, if it switches only once then it may save at most $SC + reqCost$ over the cost of processing $\beta$ entirely in $impl_1$ but incurs a cost of $SC_1$ in switching implementations. Hence, its cost is at least $l_1 + SC_1$. Switching multiple times in processing $\beta$ will not decrease its cost any further.

C: *Adv* starts with $impl_2$ active and ends with $impl_2$ active. If *Adv* does not change modes at all during this phase, then by Property 2 the cost to *Adv* is at least $l_1$ to process $\beta$. However, according to the Invariant, there is a debt of $SC_1$ to *Adv*, since it starts $\beta$ with $impl_2$ active. We remove that debt and charge it to this phase of the algorithm. Hence, *Adv* will have at least the cost of $l_1 + SC_1$. By an argument similar to Property 1, by changing its mode during the processing of this sequence, *Adv* will not decrease its cost any further.

D: *Adv* starts with $impl_2$ active and ends with $impl_1$ active. This is similar to the preceding case, except that it has an additional cost of $SC_2$ for switching to $impl_1$. Hence, its cost is at least $l_1 + SC_1 + SC_2 = l_1 + SC$.

Now consider $\gamma$. Let $r'$ be the last request in $\gamma$. Note that

$$Cost(r', impl_2) \leq Cost(r', impl_1) + reqCost$$

The two properties stated above for $\beta$ have analogs in processing $\gamma$ (we omit the proofs as they are similar to those given above).

*Property 3:* Let $\mathcal{A}$ be any algorithm that processes $\gamma$ by initially starting with $impl_2$ active, ending with $impl_2$ active, but switching between implementations any number of times in between. Then $Cost_{\gamma}^{\mathcal{A}} > Cost(\gamma, impl_2) - reqCost$.

*Property 4:* For any $\gamma_1$, $\gamma_2$ such that $\gamma = \gamma_1\gamma_2$,

$$Cost(\gamma, impl_2) < Cost(\gamma_1, impl_2) + Cost(\gamma_2, impl_1) + SC + reqCost$$

Let $l_2 = Cost(\gamma, impl_2) - SC - reqCost$. Now we use Properties 1 through 4 above and determine the

cost to any adversary *Adv* of processing $\gamma$ by examining the four cases E through H.

E: *Adv* starts with $impl_2$ active and ends with $impl_2$ active. If it never switches implementations, its cost is $l_2 + SC + reqCost$. Otherwise, say, it switches from $impl_2$ to $impl_1$ and back to $impl_2$ during processing of $\gamma$. By Property 3, this can decrease its cost by at most *reqCost*. However, since *Adv* ends the processing of segment $\omega$ with $impl_2$ active (and will start the next segment with $impl_2$ active), we remove $SC_1$ in cost from this phase of the algorithm and pay the debt, thereby maintaining the Invariant. Its costs are therefore $l_2 + SC - SC_1 = l_2 + SC_2$.

F: *Adv* starts with $impl_2$ active and ends with $impl_1$ active. By Property 4, if it only switches once, then it may save at most $SC + reqCost$ over processing $\gamma$ entirely with $impl_2$ active but incurs a cost of $SC_2$ to switch modes. Hence, its cost is at least $l_2 + SC_2$. Switching multiple times in processing $\gamma$ will not decrease its cost any further.

G: *Adv* starts in $impl_1$ mode and ends in $impl_1$ mode. If *Adv* does not change modes at all during this phase, then by Property 4 the cost to *Adv* is at least $l_2$ to process $\gamma$. By an argument similar to Property 3, by changing its mode during the processing of this sequence, *Adv* will not decrease its cost any further.

H: *Adv* starts in $impl_1$ mode and ends in $impl_2$ mode. This is similar to the preceding case, except that it has an additional cost of $SC_1$ for switching to $impl_2$. However, since *Adv* ends the iteration of the $\omega$ segment in $impl_2$, we need to remove $SC_1$ in cost from this phase of the algorithm and assign it to the debt in order to maintain the Invariant. Hence, its cost is at least $l_2$.

To analyze the overall complexity of *Adv*, we consider the cost of executing one complete segment of $\omega$. *Adv* must follow one of the following eight "paths": $A \rightarrow G$, $A \rightarrow H$, $B \rightarrow E$, $B \rightarrow F$, $C \rightarrow E$, $C \rightarrow F$, $D \rightarrow G$, $D \rightarrow H$. For instance, the first path says that *Adv* first implements case A for $\beta$ and then implements case G for $\gamma$. Note that these eight paths are the only ones possible, as A cannot be combined with E, for example, since case A states that $impl_1$ is active at the end of $\beta$, and case E states that $impl_2$ is active at the start of $\gamma$ (= end of $\beta$). By summing the costs of each of these paths, one sees that $Cost_{\beta\gamma}^{Adv} \geq l_1 + l_2 + SC$. The cost to Delta is at most $l_1 + SC + reqCost$ to process $\beta$, $SC_1$ to switch to $impl_2$ at the end of $\beta$, $l_2 + SC + reqCost$ to process $\gamma$, and $SC_2$ to switch to $impl_1$ at the end of $\gamma$. Hence,

$$\begin{aligned} Cost_{\beta\gamma}^{Delta} &\leq l_1 + SC + reqCost + l_2 + SC \\ &\quad + reqCost + SC_1 + SC_2 \\ &= l_1 + l_2 + 3*SC + 2*reqCost \\ &\leq (3 + \epsilon)*(l_1 + l_2 + SC) \\ &\leq (3 + \epsilon)*Cost_{\beta\gamma}^{Adv} \end{aligned}$$

Hence, Delta is $(3 + \epsilon)$-competitive on any $\omega$ segment.

To complete the proof of this theorem, we need to show that for any arbitrary long sequence that is a strict subsequence of an $\omega$ segment, Delta is also $(3 + \epsilon)$-competitive. We leave this as a straightforward exercise, based on the discussion above.

**End of proof**

## Lower bounds

In this section we show that there are instances of the two-implementation-component problem such that no algorithm is better than 3-competitive. The proof involves the dynamic pub/sub problem, given in the section "Examples."

In analyzing adaptive on-line algorithms, it is useful to differentiate between different types of adversaries.[4,7] An *oblivious* adversary is one that constructs the sequence of requests without regard to the actions taken by the algorithm. An *adaptive* adversary determines the action corresponding to each element of a sequence on line by taking into account the previous choices made by algorithm. If, additionally, the adversary processes the sequence only after the entire sequence has been generated, it is then said to be an *adaptive off-line* adversary. This is the most powerful sort of adversary, and we follow the terminology of Reference 8 and call it a *strong* adversary.

The result of this section shows that no deterministic or randomized algorithm can do better than Delta against a strong adversary for all two-implementation-component problems. Also, since the distinction between strong and oblivious adversaries is irrelevant for deterministic algorithms,[7] it also shows that no deterministic algorithm is better than 3-competitive against any type of adversary for *all* two-implementation-component problems.

**Lemma 1:** *Let Alg be any adaptive pub/sub algorithm controlling an individual client. There exists an arbitrarily long sequence $\alpha$ constructed by a strong adversary such that, as the length of $\alpha$ tends to infinity, Alg is at best 3-competitive on $\alpha$.*

**Proof:** Consider just two clients, $client_1$ and $client_2$, and let $client_1$ be running algorithm *Alg*. For any integer $l$, we can construct a sequence of the form $\alpha = read_1^{m_1}\ write_2^{n_1}\ read_1^{m_2}\ write_2^{n_2} \cdots read_1^{m_k}\ write_2^{n_k}$ with the following characteristics ($r_1^m$ indicates $m$ consecutive $client_1$ requests of type $r$):

- The length of the sequence (number of operations) is larger or equal to $l$
- For all $t$, $1 \leq t \leq k$, $client_1$ subscribes after the $read_1^{m_t}$ operations and unsubscribes after the $write_2^{n_t}$ operations.

To find this sequence, we begin by "serving" read requests to the $client_1$ until it subscribes and then "serving" write requests to $client_2$, causing update messages to propagate to $client_1$ until $client_1$ unsubscribes. We continue this process until we reach the desired sequence length. Since we are dealing with a strong adversary, it knows exactly when to serve read (write) requests, as it just waits until it sees $client_1$ subscribe (unsubscribe). We assume that $client_1$ forever alternates between Sub and NonSub modes if served enough write and read requests, respectively. Otherwise, this client would trivially not be competitive. Indeed, if it stayed in Sub mode, the adversary could forever feed it update messages whereas the adversary would incur no cost by staying in NonSub mode. Similarly, if it stayed in NonSub mode the adversary could forever feed it read messages and the adversary would incur no cost by staying in Sub mode.

Let $R$ be the total number of read operations and $W$ be the total number of write operations in $\alpha$. The cost to $client_1$ incurred by *Alg*, which we denote by $Cost_\alpha^{Alg}(1)$, equals $2R + W + (pk + k)$, as it costs $2R$ for $client_1$ read messages, $W$ for $client_1$ update messages ($= client_2$ write operations), $pk$ for the $k$ $client_1$ subscribe messages, and $k$ for the $k$ $client_1$ unsubscribe messages. Let $M = min\{2R, W, (pk + k)\}$. Now devise the strong adversary $\mathcal{SA}$ as follows:

- Case 1: $M = 2R$. Let $\mathcal{SA}$ stay in NonSub mode throughout the sequence. Then $Cost_\alpha^{\mathcal{SA}}(1) = 2R$. $Cost_\alpha^{Alg}(1) = 2R + W + (pk + k) \geq 3(2R) = 3 * Cost_\alpha^{\mathcal{SA}}(1)$. Hence, *Alg* is at best 3-competitive.
- Case 2: $M = pk + k$. Let $\mathcal{SA}$ subscribe directly before each set of read operations and unsubscribe directly before each set of write operations. Then $Cost_\alpha^{\mathcal{SA}}(1) = pk + k$. $Cost_\alpha^{Alg}(1) = 2R + W + (pK + k) \geq 3(pk + k) = 3 * Cost_\alpha^{\mathcal{SA}}(1)$. Hence, *Alg* is at best 3-competitive.
- Case 3: $M = W$. Let $\mathcal{SA}$ initially subscribe and then stay in Sub mode throughout the sequence. Then $Cost_\alpha^{\mathcal{SA}}(1) = W + p$. Recall that as $l \to \infty$, we can assume $W \to \infty$, otherwise the client is trivially not competitive. Hence, as $l \to \infty$, $Cost_\alpha^{\mathcal{SA}}(1) \to W$. Therefore $Cost_\alpha^{Alg}(1) = 2R + W + (pK + k) \geq 3W = 3 * Cost_\alpha^{\mathcal{SA}}(1)$, and *Alg* is at best 3-competitive.

**End of proof**

## Related work

The ideas in this paper are related to prior work in program optimization, competitive algorithms, and data replication policies.

Some of the work in program optimization discusses optimizing programs by choosing among a set of alternate implementations. For the most part, this work focuses on low-level static optimizations. For example, the early work on SETL examined how to choose the best set representation based upon program analysis.[9] Similarly the work on SQL (Structured Query Language) optimization can be viewed as statically choosing the best implementation for retrieving information from a database given a particular query.[10]

Closer to our strategy is the recent work described in Reference 11, in which the authors propose that the component writer provide multiple implementations of a component, assign costs to the different operations on each component, profile a program made up of multiple components running in a particular context, and construct an object affinity graph for this program. They then use a graph partitioning algorithm to find an optimal partitioning of the graph, thereby finding the optimal implementation for each component. Central to their scheme is the observation that the implementations at different components affect each other. Unlike algorithm Delta, their strategy optimizes a program globally, not just at the level of an individual component. On the other hand, their technique requires a more complicated methodology (profiling and the creation of an object affinity graph) and cannot adjust to dynamically changing contexts as Delta does.

Recent work on dynamic architecture description languages (see Reference 12 for a summary) aims at building software architectures that can respond adaptively to changes. Algorithms like Delta will be important in helping realize the goals of these architectures.

Our adaptive pub/sub application contrasts with previous work on data replication.[13,14] An extended version of Reference 15 gives a good overview of the literature on file allocation, file migration, and file replication problems. Although similar in some ways, our adaptive pub/sub application is closer to the work in References 16 and 17, which we discuss later in this section.

Our work draws inspiration from many competitive algorithms.[4,8,15,18,19] Although it may seem that this work could be used to solve our problem, a closer examination shows that this is not the case. For instance, one immediate idea for a competitive adaptive pub/sub algorithm, modeled after the snoopy caching algorithm,[8,18] is for a client to subscribe after performing $(1/2)p$ consecutive read operations, and to unsubscribe after receiving $p$ consecutive update operations. The asymmetry between reads and updates is due to the fact that in our model a read is twice as expensive as an update operation. Consider the sequence $(read_1^{(1/2)p-1}, write_2)^n$; that is, client$_1$ issues $(1/2)p - 1$ reads followed by a write by *client*$_2$. This is repeated $n$ times. The cost to the snoopy caching algorithm for this sequence with regard to client$_1$ would be $n * (p - 2)$. One can devise an *adversary* that would process this sequence at cost $p + n$ (the adversary would initially subscribe and then stay in Sub mode). There do not exist constants $c$ and $d$ *independent* of $p$ such that $n(p - 2) \leq c(n + p) + d$ for all $n$. Hence, the snoopy caching algorithm is not a good competitive algorithm for the pub/sub problem.

As mentioned, our work on the adaptive pub/sub problem is most closely related to the work in References 16 and 17. Reference 16 seems to exactly address our problem and to derive a better lower bound than ours ($2k$-competitive, for any integer $k$). In their model, however, the data being replicated are always of unit size, that is, variable $p - 1$ (the size of the data being replicated) always takes value 1. Hence, our work generalizes that result in an important way, by allowing arbitrary size data.

Let us see how the algorithm in Reference 16 would work for our problem. This algorithm re-evaluates the mode a client is in after every $k$ operations (alternatively, the algorithm in Reference 20 re-evaluates after a fixed time period $t$). If read operations are the majority, the client switches to (stays in) Sub mode; otherwise it switches to (stays in) NonSub mode. Consider the sequence $(read_1^k, write_2^k)^n$; that is, $k$ reads by client$_1$ are followed by $k$ writes by some other client. This is repeated $n$ times. The cost with respect to client$_1$ would be $(p + 1)n + 3kn$. One can devise an adversary whose cost would be $W = min\{nk, (p + 1)n\}$. There do not exist constants $c$ and $d$ independent of $p$ such that $(p + 1)n + 3kn \leq cW + d$ for all $n$. Because $p$ may be arbitrarily large, this may result in a large competitive factor. Hence, the algorithm would not perform well in the worst case when one generalizes to larger data structures. The algorithm does model other aspects that we have not considered. Most importantly, whereas we assume a fixed data "server," Reference 16 uses the notion of a *primary site* that can dynamically change in the course of execution.

**Dynamically switching between implementations of a component should become a useful tool in autonomic computing.**

Very recently, as this paper was going into production, I became aware of the very relevant work on metrical task systems.[21] Metrical task systems are very similar to the adaptive component problem described in this paper, and the results there are consistent with the results of this paper. However, there are two important differences between the two. First, in metrical task systems, it is assumed that $SwitchCost(i, j) = SwitchCost(j, i)$ (symmetry). We make no such assumption. As a matter of fact, for the distributed pub/sub problem described in the section "Examples," this is not the case. Second, in our model we assume that once a request is received, the processing of the request must be completed before switching to another implementation. We took this approach to better model scenarios where rapid response time is critical (i.e., the response must be made before embarking on the relatively lengthy process of switching implementations). In metrical task systems, however, switching implementations (states) can be done after a request is received but before it is processed. For both of these reasons, our work

generalizes metrical task systems in the case of two implementations (or, in the terminology of Reference 21, in the case of two states), and it is not clear that the algorithms and proofs given in metrical task systems apply directly to the adaptive component problem.

## Summary and open issues

This paper provides a framework for analyzing the effectiveness of components that switch implementations dynamically at run time based upon run-time workload characteristics. We believe that dynamic selection of component implementations will become an effective tool in optimizing component performance, and therefore the framework given in this paper should become increasingly important.

An important contribution of this paper is the near optimal $(3 + \epsilon)$-competitive algorithm Delta, for the *two-implementation-component problem*, in which there are only two implementations to choose from. An obvious question is whether there exist competitive algorithms for components with an arbitrary number of implementations. In Reference 5 we show there does not exist an algorithm that is better than $k$-competitive when the number of implementations is $k$. Another limitation of Delta is that it requires a switch from one implementation to another in "one shot," and this can often be expensive. In Reference 5 we show how Delta can switch implementations in an incremental fashion for the two example problems of section "Examples," thereby amortizing the cost of the switch operation. Still, a more general method of incremental switching between component implementations is needed.

Competitiveness is only one criterion by which to judge an algorithm for switching among implementations. In Reference 5 we mention one other criterion: convergence. Other metrics can be used as well. For instance, it may be that some domains exhibit great periodicity of patterns in event sequences. The ability to learn these patterns and then optimize to them is an important strategy in these domains.

Most importantly, we need experience to see how well Delta works in practice. This will shed light on how to best evolve this algorithm. For instance, it may be that domain knowledge needs to be taken into account in order to obtain better results.

## Cited references


1. E. Roman, S. W. Ambler, T. Jewell, and F. Marinescu, *Mastering Enterprise JavaBeans*, *2nd Edition*, John Wiley & Sons, Inc., Hoboken, NJ (December 2001).
2. S. Graham, S. Simeonov, T. Boubez, G. Daniels, D. Davis, Y. Nakamura, and R. Neyama, *Building Web Services with Java: Making Sense of XML, SOAP, WSDL and UDDI*, Sams Publishing, Indianapolis, IN (December 2001).
3. D. D. Sleator and R. E. Tarjan, "Amortized Efficiency of List Update and Paging Rules," *Communications of the ACM* **28**, No. 2, 202–208 (February 1985).
4. R. Motwani and P. Raghavan, *Randomized Algorithms*, Cambridge University Press, Cambridge, UK (1995).
5. D. Yellin, "Competitive Algorithms for the Dynamic Selection of Component Implementations," unpublished. Available from the author at dmy@us.ibm.com.
6. A. Aho, J. Hopcroft, and J. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley Publishing Co., Reading, MA (1974).
7. S. Ben-David, A. Borodin, R. M. Karp, G. Tardos, and A. Wigderson, "On the Power of Randomization in Online Algorithms," *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing*, ACM, New York (1990), pp. 379–386.
8. A. R. Karlin, M. S. Manasse, L. A. McGeoch, and S. Owicki, "Competitive Randomized Algorithms for Non-Uniform Problems," *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms*, ACM, New York (January 1990), pp. 301–309.
9. J. Schwartz, "Automatic Data Structure Choice in a Language of Very High Level," *Communications of ACM* **18**, No. 12, 722–728 (1975).
10. J. Ullman, *Principles of Database and Knowledge-Base Systems*, Volume 2, Computer Science Press, Rockville, MD (1989).
11. K. Hogstedt, D. Kimelman, V. T. Rajan, T. Roth, V. Sreedhar, M. Wegman, and N. Wang, "The Autonomic Performance Prescription," unpublished. Available from the author at wegman@us.ibm.com.
12. N. Medvidovic and R. Taylor, "A Classification and Comparison Framework for Software Architecture Description Languages," *IEEE Transactions on Software Engineering* **26**, No. 1, 70–93 (January 2000).
13. R. Ladin, B. Liskov, and L. Shrira, "A Technique for Constructing Highly Available Distributed Services," *Algorithmica* **3**, 393–420 (1988).
14. R. Alonso, D. Barbara, and H. Garcia Molina, "Data Caching Issues in an Information Retrieval System," *ACM Transactions on Database Systems* **15**, No. 3, 359–384 (1990).
15. Y. Bartal, A. Fiat, and Y. Rabani, "Competitive Algorithms for Distributed Data Management," *Proceedings of the 24th Annual ACM Symposium on the Theory of Computing* (STOC), ACM, New York (May 1992), pp. 39–50.
16. Y. Huang and O. Wolfson, "A Competitive Dynamic Data Replication Algorithm," *Proceedings of the Ninth International Conference on Data Engineering* (ICDE93), Vienna, Austria, April 1993; IEEE, New York (1993), pp. 310–317.
17. O. Wolfson and S. Jajodia, "An Algorithm for Dynamic Data Distribution," *Proceedings of the 2nd Workshop on the Management of Replicated Data* (WMRD-II), IEEE, New York (November 1992), pp. 62–65.
18. A. R. Karlin, M. S. Manasse, L. Rudolph, and D. Sleator, "Competitive Snoopy Caching," *Algorithmica* **3**, No. 1, 70–119 (1988).

19. A. Fiat, R. Karp, M. Luby, L. A. McGeoch, D. Sleator, and N. E. Yong, "Competitive Paging Algorithms," *Journal of Algorithms* **12**, No. 4, 685–699 (1991).
20. O. Wolfson, S. Jajodia, and Y. Huang, "An Adaptive Data Replication Algorithm," *ACM Transactions on Database Systems* (TODS) **22**, No. 2, 255–314 (1997).
21. A. Borodin and R. El-Yaniv, *Online Computation and Competitive Analysis*, Cambridge University Press, Cambridge, UK (1998). See Chapter 9 "Metrical Task Systems."

*Accepted for publication August 30, 2002.*

**Daniel M. Yellin** *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598 (dmy@us.ibm.com).* Dr. Yellin joined the Watson Research Center in 1987 after completing a Ph.D. program in computer science at Columbia University. He has authored over 20 journal and conference papers, as well as two computer science books, in diverse areas, which include distributed computing, program analysis, and algorithms for incremental computations. Dr. Yellin has held various positions on research and standards committees. He was editor of the International Organization for Standardization (ISO) standard on Remote Procedure Call and Vice-Chair of Operations for the Association of Computing Machinery (ACM) Special Interest Group on Programming Languages (SIGPLAN). In April 1999 he was named IBM Distinguished Engineer, and in August 1999 he was elected to the IBM Academy of Technology. He currently is Director, Software Technology Department, at IBM Research.

# Security in an autonomic computing environment

by D. M. Chess
C. C. Palmer
S. R. White

System and network security are vital parts of any autonomic computing solution. The ability of a system to react consistently and correctly to situations ranging from benign but unusual events to outright attacks is key to the achievement of the goals of self-protection, self-healing, and self-optimization. Because they are often built around the interconnection of elements from different administrative domains, autonomic systems raise additional security challenges, including the establishment of a trustworthy system identity, automatically handling changes in system configuration and interconnections, and greatly increased configuration complexity. On the other hand, the techniques of autonomic computing offer the promise of making systems more secure, by effectively and automatically enforcing high-level security policies. In this paper, we discuss these and other security and privacy challenges posed by autonomic systems and provide some recommendations for how these challenges may be met.

As computing systems have become more complex, more interconnected, and more tightly woven into the fabric of our lives, the resources involved in managing and administering them have grown at a steadily increasing rate. As the costs of system hardware and software have leveled off or decreased, the costs of the human resources devoted to system administration have continued to grow, and therefore constitute a steadily larger fraction of information technology (IT) costs. The autonomic computing initiative is aimed at addressing these increasing costs by producing computing systems that require less human effort to administer; systems that, like the biological systems that keep our hearts beating and our body chemistry balanced, can take care of routine and even exceptional functions without human intervention.[1]

Like any other significant computing system, autonomic systems need to be secure. Building secure autonomic systems is a challenge for a number of reasons. Many autonomic systems will use new techniques and new architectures whose security implications are not yet well understood. Autonomic systems should not rely on anomalous behavior caused by security compromises being noticed by humans, if they are to benefit from reduced human administration costs. Because many autonomic systems are expected to deal with a constantly changing set of other systems as suppliers, customers, and partners, they need flexible new methods for reliably establishing trust, detecting attacks and compromise, and recovering from security incidents. Because some autonomic systems deal with personal information about individuals, they need to be able to represent and demonstrably obey privacy policies required by national laws and business ethics.

Successful autonomic systems will need to be self-configuring, self-optimizing, self-protecting, and self-

healing. Although security concerns are most obvious in protecting the system from attack and in recovering from the effects of attacks, security will be key in all other aspects as well. Systems must be secure in every configuration into which they might put themselves, and in every state into which they might optimize themselves. Systems must be robust against attempts to provide them with false or misleading information that might lead them to configure or optimize themselves insecurely, to enter into an unjustified trust relationship, or to fail to protect adequately against a malicious attack.

Autonomic computing will not reinvent computer science *ex nihilo* and the security of autonomic systems will not be an entirely new kind of security. All the traditional issues familiar to computer security researchers will arise in autonomic systems, some in more complex and urgent forms. And just as widespread program sharing and ubiquitous network connectivity took computer viruses and worms from a theoretical possibility to an annoying oddity and then to a major security concern, we should expect that the new computing environments made possible by autonomic computing will give rise to unique security threats of their own.

At least as significantly, the new abilities offered by autonomic computing will also include ways to make our systems more secure and our private data better protected. Building and administering secure computing systems is well known to be a difficult task; properly configuring a complex system to conform to security policies specified at a high level is extremely challenging even for skilled practitioners, and even the best-administered computing systems generally conform only approximately to their putative security policies.[2] By automating the process of configuring, optimizing, and protecting systems according to explicitly stated security policies, autonomic systems offer us the opportunity to do better.

In the next section of this paper, we provide a very brief overview of some of the architectural features that will be important in the design of autonomic computing systems, with an emphasis on the aspects of that architecture that relate to security. In successive sections, we survey a number of old and new security issues and opportunities as they apply to autonomic systems, and we describe two existing systems in which some of these issues have begun to emerge. Along the way, we will note both the challenges and the opportunities in providing security for autonomic computing systems.

## Architectural features of autonomic computing

Autonomic computing will have implications for computing systems at all scales, from single devices to the worldwide networked economy. At small scales, we anticipate that the units of autonomic computing, generally referred to as "autonomic elements," will be comparatively simple and of fixed function, performing the same activities in concert with the same set of other elements for long periods of time. At higher levels, however, we expect that many autonomic elements will function in a very dynamic environment, in which only the element's essential mission and governing policies will remain constant. The details of how they carry out their mission and what other elements they interact with may change every day, or even every second.

We anticipate that one very common architecture for an autonomic element will involve two parts: a *functional unit* that performs whatever basic function the element provides (such as storage, database functions, Web services, and so on), and a *management unit* that oversees the operation of the functional unit, ensures that it has the resources that it needs to perform its function, configures and reconfigures it to adapt to changing conditions, carries out negotiations with other autonomic elements, and so on.[3] Figure 1 shows a simple conceptual diagram of an autonomic element consisting of a functional unit and a management unit.

The thin arrows connecting the management unit to the world outside the autonomic element represent the management unit's dealings with other autonomic elements (and potentially with other external resources). The thick arrows connecting the functional unit to the outside world represent the channels by which the element acquires the resources that it needs to carry out its basic function, and by which it delivers the results of that function to other elements. The arrows between the management unit and the functional unit represent the sensors and effectors by which the management unit monitors and controls the functional unit, and the arrows between the management unit and the loops around the function arrows represent the access control that the management unit exercises over these functional channels. No autonomic element or other entity can provide a resource to this element, or obtain any ser-

vice from it, without the permission of the management unit, as negotiated and obtained through the management channels.

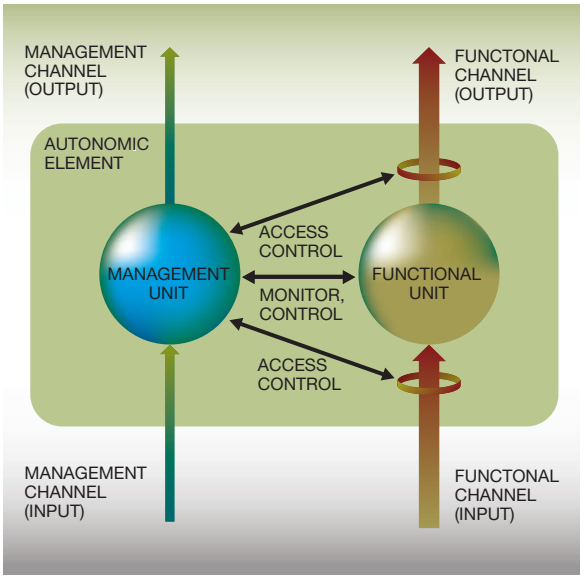
In order to make the decisions necessary to properly oversee the operation of the functional unit and to achieve the flexibility required to make the element self-managing, many management units will carry with them, or otherwise have access to, *policies* that govern and constrain their behaviors at a comparatively high level and *task and state representations* that functionally describe their current mission, strategy, and status at a lower level. Unlike conventional computing systems, which behave as they do simply because they are explicitly programmed that way, the management unit of an autonomic element will often have a wide range of possible strategies available to it in fulfilling the policies that govern it, and an explicit representation of the current state of its efforts to carry out those policies.

Some of the policies that govern an autonomic element will be security policies. An element's security policies may include descriptions of what level of protection needs to be applied to the various information resources that the element contains or controls, rules that determine how much trust the element places in other elements with which it communicates, what cryptographic protocols the element should use in various situations, under what circumstances the element should apply or accept security-related patches or other updates to its own software, and so on. Other policies will control the strategies that an element uses to recover when one of its suppliers fails to provide an expected resource, and to which of its commitments to give the highest priority when not all can be fully met. These policies will either be directly specified by a human, implicitly specified (as by a human accepting a default), or derived from higher-level policies by the rules of the appropriate policy calculus.

Some of the task and state representations that a management unit holds to describe the current status and activities of the element will also be relevant to the element's security. A management unit may, for instance, have:

- A representation of the other elements upon which it currently depends, and how much it trusts each of them
- A representation of the current life-cycle state of the software that the element is running and whether or not there are any security updates available for it
- A list of contact information for one or more other autonomic elements or human administrators who should be notified when certain suspicious circumstances are observed
- Agreements with one or more other autonomic elements to provide it with security-relevant information, such as log-file analyses or secure time-stamping
- A list of previously-vetted resource suppliers, used to quickly verify the digital signatures on the resources they provide

Figure 1 Logical structure of an autonomic element




By explicitly representing both security policies and security-related tasks and states within the system, autonomic elements will be able to automatically handle a wide range of security issues that are currently addressed by human intervention or by comparatively *ad hoc* programmed solutions. Although we do not expect that every autonomic element at every scale will fully conform to this model of management and functional units, policies, and representations, we do anticipate that these design patterns will underlie virtually all autonomic systems above the device level, and they form the conceptual framework for our consideration of the security aspects of autonomic systems.

## Traditional and emergent security issues

Every computer security issue familiar to workers in the field will be relevant to autonomic computing. Autonomic elements will need secure authentication protocols to identify each other, secure cryptography to keep their communications from falling into the wrong hands, and secure delegation mechanisms to allow other elements to take actions on their behalf. Digital signatures and nonrepudiation mechanisms will be vital for elements that carry out electronic commerce, and it will be crucial to ensure that the computers on which autonomic systems run are not vulnerable to compromise through buffer overflows, or to service disruption by network flooding. Automatic intrusion detection systems will be more important than ever, since in the absence of direct human control over the operations of the autonomic elements, it will not be possible (or at least not desirable) to rely on human common sense to intuit that something is not right.

The new security issues that emerged in the last years of the twentieth century will continue to apply in autonomic systems. Replicating threats such as computer viruses and worms, and subtler network effects such as routing and feedback loops, will need to be detected and eliminated in autonomic systems at least as urgently as they are in today's systems. A worm that could infect and proliferate between autonomic elements based on a particular implementation library could potentially be at least as severe as a worm that spreads between computers running a particular HyperText Transfer Protocol (HTTP) server.[4] Distributed denial-of-service attacks,[5] in which a large number of systems are compromised remotely and then used in a coordinated attack on a target, would be as devastating in an autonomic system as they are today, if not more so.

The security issues that are urgent today will be even more urgent in a world of autonomic systems. That new world will also bring new security issues of its own, issues that may not be significant or present at all today. At the same time, autonomic technology will offer new opportunities—new ways of securing our systems against attack.

## Control, information, and trust

When a system is within a given administrative domain, the owners of that domain have (at least potentially) full control over that system, and complete information about it. In contrast, the owners of a domain have less control over, and less complete and reliable information about, systems in other domains. All other things being equal, a given party will have less trust in systems that are outside its administrative domain, because it can neither completely control, nor completely know, what those systems are doing.

In a dynamic autonomic computing system that spans multiple domains, no single entity has full control over, or full knowledge of, all the components of the system. Each party using the system will therefore have a reduced degree of trust in the system. To ensure that such a system is usable, and used, by the parties for whom it is intended, we must design the system to increase the level of trust that those parties have in the system, so that the net loss of trust is small compared to the gain in efficiency and flexibility from using the autonomic system.

In the current world, businesses and individuals are accustomed to dealing with entities that they neither entirely control nor perfectly trust, as customers, as suppliers, and as partners. In constructing autonomic computing systems that span administrative domains, we must allow businesses and individuals to operate successfully in an environment where critical parts of the IT infrastructure, and of the business and personal infrastructure in general, are similarly under the control of others, and where the details of the relationships with those others can change without human intervention.

Trust is necessary both for the routine operation of autonomic computing systems and for the initial adoption of those systems by customers and users. Not only must the elements of an autonomic system have good reason to trust the other elements that they discover and with which they interoperate, but the human decision makers who opt to use an autonomic system in the first place must have good reason to trust that the system as a whole will serve their purposes. Establishing both of these kinds of trust will require considerable invention. Human decision makers will always prefer to make their own trust judgments in some areas. On the other hand, automated trust establishment in some domains may come to be recognized as more reliable and consistent than *ad hoc* manual trust decisions made by humans.

There are a number of existing mechanisms for establishing and reasoning about trust and trustworthiness, and some of them will have a role to play

in the development of trust in autonomic systems. These range from the decentralized web of trust used in PGP-style (Pretty Good Privacy-style) systems[6] to the strict hierarchy of certificate authorities used in others.[7] An autonomic element might make trust decisions based on a particular trust scheme, depending either on explicit instructions from the humans controlling it, or by a policy derived through the appropriate policy calculus from higher-level human-provided policies.

When one autonomic element relies on information that it gets from another, it should ideally have access to the entire chain of elements that produced that information. On the other hand, if the elements involved are not all in the same administrative domain and do not all have precisely the same goals, the element supplying the information may not be willing (that is, may not be permitted by its own security policies) to reveal the identities of all the other elements in that chain. In such circumstances, how can the element receiving the information determine to what extent the information can be trusted? There will be roles here for technical methods (such as zero-knowledge proofs), for explicit rules (where, say, a human simply reassures the element that certain information sources can be relied upon), for contracts and other legal arrangements (where an element can assume that information can be trusted, because it knows that legal recourse exists if it turns out to be false), and for various kinds of trust calculus.

The security and trust policies that govern an autonomic element will determine how demanding it is when making trust decisions, how readily it trusts other elements, and how much corroboration it seeks before relying on information. These policies will also constrain some of the actions that an element can take. For instance, if one of an element's suppliers becomes unavailable or stops performing acceptably, the element will need to decide which of the potential replacement suppliers can be trusted, both to take over the function of the failed supplier competently, and to be sufficiently trustworthy in that role. By enabling computing systems to make these decisions in consistent and reliable ways, autonomic techniques will engender an extremely adaptive and dynamic operational style, without compromising security.

## System compromise vs system outage—coping with intrusions

Autonomic systems will be self-protecting and self-healing. This means, in part, that they will be capable of detecting intrusions on their own, and reacting to them so as to eliminate the intrusion and restore the system to an uncompromised state. To examine how this might be done, we divide an intrusion into three phases:

- Compromise—the initial intrusion and subsequent actions by the intruder
- Detection—the determination, by the system, that an intrusion has occurred
- Restoration—the elimination of the intrusion and restoration of the system to an uncompromised state

In many ways, the compromise of a system by an intruder presents a reliability problem. The system may or may not still function, and it is likely to function in a different and unintended way. But compromise is also different from ordinary unreliability in important ways. In an unreliable system, we generally assume that errors are uncorrelated, in the sense that having two simultaneous errors is much less likely than having one, or none. Compromise can easily violate these assumptions. Attacks are malicious rather than random, and the compromise of several parts of the system can be correlated to further the purposes of the attacker. On the other hand, while some system faults are detectable precisely because of the pattern of failures that they produce, human attackers are motivated to cover their tracks and keep their intrusions inconspicuous. These factors make intrusion potentially more dangerous, and more difficult to detect and cure, than simple reliability problems. Some of the means that an autonomic element uses to deal with an intrusion will be similar to the means it uses to deal with unreliable hardware, or other nonmalicious failures, but other means will be very different.

Systems are usually organized so as to prevent intrusion. Tools such as firewalls, passwords, and access control limit the ability of external parties to act upon a system. At one time, preventative measures were thought to be the solution to the problem.[8] It seems likely, however, that additional protective measures are necessary, both because it is difficult to deploy a fully secure system, and because inevitable design and implementation flaws provide targets of attack even in supposedly secure systems. Fine-grained behavioral restrictions have been used to further limit what a user can do within a system by limiting what actions various programs can take (see for instance the signer-based security model in Java** 2[9]). These can be useful, but they too have

their limitations (for instance, specifying precisely which behaviors should be permitted for a given program can be a complex and error-prone task). Finally, detection systems attempt to notice when an intrusion is taking place, often by noticing actions that should never occur in a normal system, or by noticing when a user's or a program's actions go beyond some statistical measure of normal operation.[10]

As with the detection of virtually any behavioral characteristic of a system, it is impossible to detect with 100 percent certainty that a system has been compromised. Therefore, detecting an intrusion or a compromise in an autonomic system will always be an approximate business. Even the best detection methods will always have false positives or false negatives, or both. Because of the malicious nature of an intrusion, we must assume that it occurs before we are aware of it, leaving the intruder some amount of time to compromise the system. The best an autonomic system can do is to minimize the amount of time between the intrusion and its detection, so as to limit the amount and extent of compromise.

Once we are aware that an intrusion has taken place and have some idea of what parts of the system may have been compromised, we can act. The first thing we must do is to cut off the intruder as much as possible. External communications links to the affected parts of the system can be severed, and affected machines can be taken down. Autonomic elements that have been compromised can be terminated or isolated through changes to high-level policies. If this stops the attack, our remaining problem is restoring the system to an uncompromised state. To do this, we divide the system conceptually into a stateless part and a stateful part. The stateless part has no important state that changes as the system operates, so restoring it is relatively easy. We can restore each system from a known secure backup (perhaps the original distribution set for the system) and bring it up again. Autonomic systems must be able to perform these functions automatically (at least in typical cases), without human intervention.

Restoring the stateful part of the system requires more preparation. The state of the system must be maintained in such a way that (1) corruption can be detected and (2) corruption can be eliminated. One approach to these requirements involves keeping the state of the system in an encrypted, redundant form, distributed across a number of logical and/or physical nodes. Techniques are available for restoring state where as much as one-third to one-half of the copies of the state have been corrupted (see the section "Secure distributed storage," later). In many cases, it will be advantageous to distribute redundant copies of the state in any case and check them against each other periodically, so that random errors can be dealt with. Using encryption and keeping additional copies helps us deal with intrusion in a similarly general way; this is one technique that autonomic elements can use against both malicious and nonmalicious failures.

Once the compromised systems have been restored, they can be brought back on line, their communication channels can be restored, and they can once again operate normally. These operations are currently performed by skilled human operators; automating them so that autonomic systems can perform them automatically in typical cases (requiring human help only rarely) will be a significant challenge.

### Fraud and persuasion

The policies that govern an autonomic element's high-level behavior, and the task and state representations that allow it to reason about its own activities, provide high-value targets to a potential attacker. When an attacker compromises a traditional computing system, the attacker may, for instance, insert a piece of code that causes the system to silently send him or her a copy of some important information at a particular e-mail address at a particular time. If that address becomes unavailable, or a network gateway blocks the transmission of the information, the leak will stop. But if an attacker were to compromise an autonomic element, and add to its policy database a policy that required it to provide some important information to the attacker at a certain interval, the autonomic element would then use every resource at its disposal to ensure that the information was delivered. The attacker would have harnessed the element's own ability to adapt to changing conditions and adopt new strategies for the purpose of stealing the desired information. Preventing this sort of high-level subversion will be an important part of the security of autonomic systems.

On the other hand, the security policies that govern an autonomic element can, if properly secured against tampering, provide new levels of resistance to attack. If the attacker in the example above alters only the functional code of the element, or only the task representation held by the management unit, the element will probably not leak its important information to the attacker, because that leakage will

be forbidden by its security policy. Autonomic systems, because they contain explicit computer-readable representations of the security policies under which they operate, can potentially be more resistant to attack and subversion than current systems, which contain only functional code whose behavior may or may not conform to the policies to which humans would like it to conform.

In a preautonomic system (see Figure 2), an attacker who can implant a back door into a computing system can create a data leak that will extract and send any information the back door code can identify. In an autonomic system (see Figure 3), even if an attacker can implant a back door into the programming of the functional unit, the element's management unit will typically block the back door code's attempt to leak data back to the attacker, because the element's security policies will not allow the transmission.

Autonomic elements will make many decisions without direct human instruction, and they will automatically sense and adapt themselves to changes in their operating environment. In these respects, the field of autonomic computing shares features with the only slightly older field of autonomous agents (see for instance Reference 11). Like autonomous agents, autonomic elements will depend for their correct operation on accurate information from other elements and from the outside world. Although there has been some theoretical work on protecting autonomous agents from attacks based on providing them with inaccurate or biased information (see for example Reference 12), this is still a mostly unsolved problem; autonomic elements will have to include safeguards to prevent such deception. In the short term, these safeguards will likely work primarily through policies that instruct the elements to rely only on information derived from sources that a human has explicitly declared trustworthy. As policies become more complex and more flexible, more of the work of making trust decisions can be put into the elements themselves. In fact, this will quickly become a necessity, as the complexity of the system of autonomic elements grows.

Figure 2 Data leak via back door implanted in functional unit

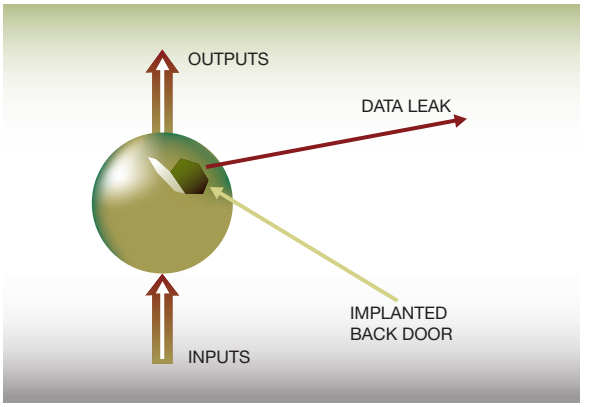
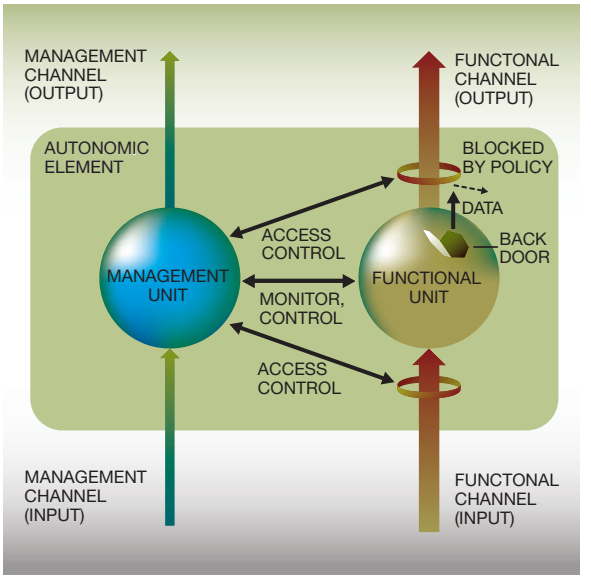



Figure 3 Blockage of back door leak by autonomic system




## Privacy issues

One of the factors in the increasing number and complexity of information systems is the increasing amount of information available to be processed. Much of this information is personal information, relevant to some individual who may desire to control how the information is gathered, and how it is used or distributed. Different jurisdictions have different definitions of personal information, give individuals different degrees of legal control over such information, and impose different legal requirements on entities that hold such information. The European Union's Directive on the Protection of Personal Data, the United Kingdom's Data Protection Act of 1988, and U.S. regulations on the privacy of medical records are current examples. Some industry groups

adhere to voluntary guidelines on personal data protection, and individual enterprises often have their own privacy policies that govern the use and distribution of personal information. Contractual relationships may also call for specific handling of personal data.

If autonomic computing is to reach its potential, autonomic elements must be able to gather, transmit, and process personal data in a way that respects the relevant privacy policies, without requiring direct human intervention or assistance. Architecturally, this means that autonomic elements must maintain representations of both the applicable data protection policies and the privacy status of the various kinds of information that they process. It must be possible for an autonomic element to reliably and automatically determine the data protection class of each unit of data that it deals with, and to securely and automatically retrieve the correct policy to apply to that class of data. It must be possible to make routine changes in policies without changing the underlying programming or architecture, since laws and guidelines on personal data protection are in a state of constant flux and are likely to remain so for some time.

**Autonomic techniques offer opportunities for increasing system security.**

## Security and privacy negotiation

It is not enough for an element of an autonomic computing system to ensure its own security or its own privacy measures. Because it depends critically on other elements, it must also be able to trust the security and privacy of those other elements. An autonomic computing system that spans domain boundaries must allow elements to negotiate security and privacy policies, and to gather and securely exchange the information required to verify compliance and to satisfy audit requirements. This will require ontologies and standards expressive enough to allow the specification and negotiation of security and privacy policies, as well as both technologies and standards for record keeping and auditing of behavior relevant to these policies. (Existing technologies in this space, such as Platform for Privacy Preferences (P3P),[13] are currently aimed at allowing individual retail users to specify simple static privacy policies; the requirements for autonomic computing will go far beyond this.) New cryptographic techniques and other technologies will be needed to allow data to be anonymized or aggregated where required, and to be encoded so as to be secure against various new and old types of threat.

Security and privacy policies and their negotiation must be able to take complex political and geographical situations into account. This is one area where geography does not disappear in cyberspace. If an autonomic element has data that, by law, may not enter a certain region of the world, then the elements to which it provides those data must be able to promise, and later perhaps auditably prove, that they did not ship the data to that region, and that each other element to which they provided those data made the same promise and can provide the same audit data. The same considerations apply to nongeographic restrictions. Powerful and flexible ways to specify how data may and may not be processed are needed, as data move through a complex autonomic system. In some applications, it may be necessary for one party to monitor, in more or less real time, the things that have happened to the sensitive data that have been provided to the system. Enabling this in a real system will be a significant challenge.

## Further opportunities

Autonomic techniques offer opportunities for increasing system security, both at the level of single elements, and in systems made up of many elements. The key advantage at the level of the single element will be the explicit representation and enforcement of security policies; as we have pointed out previously, system security will no longer rely on high-level security policies being accurately converted into low-level implementation when the system is first coded and configured. Instead, autonomic elements will maintain representations of both the security policies in effect and the tasks to be performed, and will be able to ensure that the actions they take conform to the relevant security policies. With this opportunity come challenges: it will require considerable invention to ensure that the various security policies in use by the autonomic elements in a system are compatible, to verify that the collective behavior that they produce has desirable overall security properties, and to design mechanisms to automatically handle policy conflicts that arise.

Very few autonomic elements will operate in a vacuum. Many groups of elements in an autonomic system will in fact have the same interests, or at least significantly overlapping interests. When a large number of elements have an interest in the security of some part of the system, they may be willing to provide or exchange security-relevant information and to allow it to be aggregated. An element could provide security analysis as a service; any other element willing to provide it with security-relevant data could benefit from analysis of all of the aggregated data, potentially discovering nascent or ongoing attacks that would not be visible from examining only a single data set, or that might not have been distinguishable from innocent traffic. This sort of collaborative distributed intrusion detection is sometimes done on an *ad hoc* basis today; autonomic computing will allow it to become routine. Many other security-relevant services, such as third-party signing, time stamping, and security auditing might be provided as services by elements of an autonomic system.

As presented in the next section, the practicality of automated security response has already been demonstrated in the form of "immune system"-style countermeasures against replicating threats such as viruses and worms. Autonomic computing will provide a sound infrastructural basis for this kind of automated security collaboration.

The policy-management advantages that autonomic computing offers to a single element also apply to groups of elements in the same administrative domain, or otherwise under the control of the same set of policies. Ultimately, the owner of an autonomic system will need to be concerned only with security policies at the level of overall business objectives and operating standards, and the autonomic elements comprising the system will automatically derive and distribute the lower-level policies constraining their detailed operation. In the shorter term, humans will still be involved in the creation of middle-level and even low-level policies, but by securely distributing these policies to all the elements of the autonomic system, significant amounts of effort (and opportunities for error) will be avoided.

## Sample products and implications

In this section, we discuss security issues in two existing systems that exhibit various kinds of autonomic behavior.

**Immune system.** In the 1990s, replicating security threats in the form of computer viruses and worms became a significant problem for the computing community. One effective response to that threat was the development of a biologically inspired "immune system," as described in Reference 14. Although this immune system predates the idea of autonomic computing as we present it here, it was one of the progenitors of that idea, and the architecture and design of the system have autonomic aspects that are worth considering.

Key to the effectiveness of the antivirus immune system is that it automates many of the activities involved in discovering a new virus and protecting computer systems against it. Until the development of the immune system, many parts of the process depended on human action. A new virus (discovered either by heuristics in antivirus programs or by a user noticing odd system behavior) would be manually captured and forwarded to an antivirus company. There it would be analyzed by a human expert using various debugging and analysis tools. That expert would then develop a virus "definition," containing instructions for detecting and removing the new virus. That definition would be manually added to the set of definitions used by the antivirus program, and the new definitions placed on update servers for users of the antivirus program to download. Although some antivirus programs had begun to include scheduled update facilities that would automatically download and install the latest available definitions at some fixed interval, the rest of the process was almost entirely manual, and therefore limited in speed by the availability of skilled humans.

The immune system automates every stage of this process. The antivirus software on a protected client system uses a variety of heuristic methods to detect and identify files or other objects that may contain a new virus. A suspect file is encrypted and securely transmitted to an analysis center, where it is exercised and encouraged to spread within a protected environment. After it spreads, it is automatically analyzed, new detection and repair information is extracted, and the updated definitions are tested and provided to both the original infected system and to any other systems that are registered to receive automatic updates. At various stages of the process (especially during analysis) the system will defer to human experts if the virus does not yield to automatic methods, but for a significant percentage of new viruses the entire process proceeds without human intervention.

Various aspects of the immune system contain lessons for the development of secure autonomic elements. Perhaps the most important is that in a highly connected world, security threats can spread very quickly, and it is vital that the security response be correspondingly quick. Autonomic elements concerned with security must also be able to function under the sudden heavy loads often caused by widespread security incidents; the immune system uses a hierarchical network of gateways that cache the results of virus analysis and thus prevent a sudden flood of suspect files from bogging down the analysis center.

**Autonomic elements must be able to function under the sudden heavy loads often caused by widespread security incidents.**

Autonomic elements should also have the ability to ask for human confirmation of security-relevant actions. The component of the immune system that forwards a suspect file from a protected machine to the central analysis center can be configured to ask an administrator for confirmation before sending. At the same time, it should also be possible to configure the system so that this confirmation is not required; where a sufficient trust relationship exists between the protected enterprise and the owner of the analysis center, the confirmation can be turned off, so that suspect files are immediately forwarded. The security policies that govern autonomic elements must be flexible enough to allow this sort of configurability in connection with human confirmation.

Another lesson learned from the development and deployment of the antivirus immune system is a psychological one. While the system was being developed, considerable skepticism was expressed by experts in the field (including some of the antivirus experts involved in the development of the system) about the feasibility of creating an automatic system that would perform as well as a skilled human at critical security-related tasks (such as deriving recognition definitions for computer viruses). Those fears proved unfounded. Although the system does have decision points at which it can defer to a human expert when a particular virus is not automatically analyzable, for a typical virus the system does as well or better than human experts. In particular, the virus recognition patterns extracted by the immune system proved to be more powerful for detection, and less prone to false alarms, than those manually chosen by human experts.

The antivirus immune system we have just described is not the only computer security system inspired by biological analogies; see for instance Reference 15 for another approach inspired by biological immune systems and Reference 16 for a security system inspired by homeostatic processes.

**Secure distributed storage.** The central idea of the Secure Distributed Storage (SDS) solution is to efficiently spread important information over several separate servers in such a way that it is highly available, reliable, self-correcting, and self-protecting.[17] Let us assume that we want to store a file in an SDS system. The SDS system does not simply send a copy of the file to each server, since that would require every server to have enough storage to hold a complete copy of the file. Instead, the file is processed in a special way that breaks up the file into pieces, with each piece being significantly smaller than the original file. Each piece is then sent to a different server. This provides very efficient distributed storage.

Because of the special process that was used to break the file into pieces, one has only to retrieve the original pieces from just over half of the servers in order to completely reassemble the original file. It does not matter which pieces are retrieved, as long as at least half of them are. This allows for the other servers to have gone off-line or otherwise be unavailable. In addition, the SDS also provides a means by which compromised, malicious servers may be detected when a retrieval operation is underway. These capabilities allow the SDS to provide highly available, self-healing storage.

An SDS system consists, by definition, of several distinct servers. When depositing a file into an SDS system, the user can initiate the deposit of the file via any one of these servers. That is, any one of the available servers in an SDS system can act as the user's "gateway" for storing files. Similarly, when requesting a retrieval, the user may direct the request to any available server of the SDS. This also eliminates the need for the user's system to contact each of the distributed servers when attempting a deposit or retrieval. This transparency among the servers (auto-

nomic elements) directly supports more autonomic behavior.

Finally, an SDS system can be configured to be self-protecting. That is, since any of the servers might be down, disconnected from the network, or compromised by an attacker at any time, there must be some means for reestablishing the trust of that server. If a server was unavailable or disconnected, when it becomes available once again it sends the other servers a "What did I miss?" message and they all cooperate to bring that server up to date. Similarly, if a server has been compromised, that condition can be detected, and that server is automatically ignored until it can be reinitialized to restore its trustworthiness. In addition, the SDS system is capable of changing the representation of the data that it holds, so that information "captured" by an attacker who has broken into the system is useless. This is achieved by having the system periodically shuffle the pieces of a stored file across all the servers. This proactive security feature provides the self-protecting attribute of this autonomic system.

## Conclusion

No functioning system is perfectly secure, and autonomic systems will be no exception. The development of autonomic systems cannot be delayed until we have found final solutions to all the corresponding security challenges, since such final solutions will never be available. On the other hand, for autonomic computing to succeed, it must be, and must be perceived as being, secure enough that its benefits outweigh the risks. In this paper, we have outlined a number of security and privacy challenges facing those designing and developing autonomic systems, and also a number of ways that autonomic principles can be used to make systems more secure than they are today.

There is a need for further research in many areas. Some of the key needs identified above include:

- Ways to represent and reason about the security and privacy policies that govern autonomic systems
- Ways to represent and reason about security states, and the trust relationships between elements
- Criteria and methods for effectively differentiating between normal system failures and failures caused by malicious attacks
- Policies and algorithms for making autonomic elements that are resistant to fraud and persuasion
- Common languages and taxonomies for communicating and negotiating about security and privacy states and policies
- Ways to construct individual autonomic elements so that their collective behavior is both trustworthy and trusted

Autonomic computing offers as least as many benefits in the security area as it does challenges. The complexity of modern computing systems makes secure systems administration a daunting task and one that is seldom done well in practice. Recent advances, including the growing use of automatic intrusion detection systems, secure embedded processors, proactive security measures, and automated virus response, have helped take some of the burden of security maintenance off overloaded system administrators, but there is much more to do. By making computing systems directly aware of the security policies that apply to them, and giving the systems the ability to conform their actions to those policies, the techniques of autonomic computing will help create systems that are increasingly and consistently secure.

**Trademark or registered trademark of Sun Microsystems, Inc.

## Cited references


1. P. Horn, *Autonomic Computing: IBM's Perspective on the State of Information Technology*, IBM Corporation (October 15, 2001); available at http://www.research.ibm.com/autonomic/manifesto/autonomic_computing.pdf.
2. C. C. Palmer, "Ethical Hacking," *IBM Systems Journal* **40**, No. 3, 769–780 (2001).
3. "The Dawning of the Autonomic Computing Era," by A. G. Ganek and T. A. Corbi, *IBM Systems Journal* **42**, No. 1, 5–18 (2003, this issue).
4. CERT® Advisory CA-2001-19: "Code Red" Worm Exploiting Buffer Overflow in IIS Indexing Service DLL, CERT Coordination Center (2001); see http://www.cert.org/advisories/CA-2001-19.html.
5. D. Dittrich, "DDoS: Is There Really a Threat?," Usenix Security Symposium 2000; see http://staff.washington.edu/dittrich/talks/sec2000/.
6. P. Zimmerman, *PGP User's Guide*, MIT Press, Cambridge, MA (1994).
7. "Internet X.509 Public Key Infrastructure Certificate and CRL Profile," Internet standards track protocol document, RFC2459; see http://www.rfc-editor.org/rfc/rfc2459.txt.
8. United States Department of Defense, *Trusted Computer System Evaluation Criteria*, Technical Report DoD 5200.28-STD (1985).
9. L. Gong, *Inside Java 2 Platform Security*, Addison-Wesley Publishing Co., Boston, MA (1999).
10. M. Gerken, "Statistical-Based Intrusion Detection"; see http://www.sei.cmu.edu/str/descriptions/sbid.html (1997).
11. *Proceedings of the International Conference on Autonomous Agents*, ACM, Montreal, Canada (2001). See http://autonomousagents.org/.
12. "Special Issue on Deception, Fraud, and Trust in Agent So-

cieties," C. Castelfranchi et al. Editors, *Applied Artificial Intelligence Journal* **14**, No. 8 (2000).
13. World Wide Web Consortium, Platform for Privacy Preferences (P3P) Project (2002); see http://www.w3.org/P3P/.
14. S. R. White, M. Swimmer, E. Pring, W. Arnold, D. Chess, and J. F. Morar, "Anatomy of a Commercial-Grade Immune System," *Proceedings of the Ninth International Virus Bulletin Conference* (1999).
15. S. Hofmeyr and S. Forrest, "Architecture for an Artificial Immune System," *Evolutionary Computation* **7**, No. 1, 1289–1296 (2000).
16. A. Somayaji and S. Forrest, "Automated Response Using System-Call Delays," *Usenix Security Symposium* (2000).
17. J. Garay, R. Gennaro, C. Jutla, and T. Rabin, "Secure Distributed Storage and Retrieval," *Theoretical Computer Science* **243**, No. 1–2, 363–389 (2000).

*Accepted for publication October 21, 2002.*

**David M. Chess** *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598 (electronic mail: chess@us.ibm.com).* Mr. Chess joined IBM at the Thomas J. Watson Research Center in 1981. He has worked on mainframe performance management, computer conferencing, workstation technology, computer security, and virus prevention. He was on the team that developed and supported IBM AntiVirus, and he is currently a research staff member in the Massively Distributed Systems group, working on emergent security problems and security for active content and autonomic systems, as well as the architecture of autonomic computing elements. He has an A.B. degree in philosophy from Princeton University, and an M.S. degree in computer science from Pace University.

**Charles C. Palmer** *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598 (electronic mail: ccpalmer@us.ibm.com).* Dr. Palmer is senior manager of the Network Security and Cryptography departments at the Thomas J. Watson Research Center. His teams work in the areas of cryptography research, Internet security technologies, Java security, Web services, privacy, and in the Global Security Analysis lab (where they are known as the "ethical hackers"), which he cofounded in 1995. Dr. Palmer frequently speaks on the topics of computer and network security at conferences around the world. He received a Ph.D. degree from Polytechnic University in computer science in 1994, where he was also an adjunct professor of computer science from 1993 to 1997. He holds five patents and has several publications from his work at IBM and Polytechnic.

**Steve R. White** *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598 (electronic mail: srwhite@us.ibm.com).* Dr. White received a Ph.D. from the University of California at San Diego in theoretical physics in 1982. He accepted a postdoctoral fellowship at IBM Research, where he later became a research staff member. He has published in the fields of condensed matter physics, optimization by simulated annealing, software protection, computer security, computer viruses, and information economies. He holds roughly a dozen patents in related fields. He was elected to the IBM Academy, which advises the company on technology policy and direction, and has received IBM's highest technical award for his work. At the IBM Thomas J. Watson Research Center, he is currently senior manager of the Autonomic Computing group, which explores how we can build computing systems that have billions of components and are nevertheless self-managing.

# A system model for dynamically reconfigurable software

by K. Whisnant
Z. T. Kalbarczyk
R. K. Iyer

The ability to reconfigure software is useful for a variety of reasons, including adapting applications to changing environments, performing on-line software upgrades, and extending base application functionality with additional nonfunctional services. Reconfiguring distributed applications, however, can be difficult in practice because of the dependencies that exist among the processes in the system. This paper formally describes a model for capturing the structure and run-time behavior of a distributed system. The structure is defined by a set of elements containing the state variables in the system. The run-time behavior is defined by threads that execute atomic actions called operations. Operations invoke code blocks to bring about state changes in the system, and these state changes are confined to a single element and thread. By creating input/output signatures based upon the variable access patterns of the code blocks, dataflow dependencies among operations can be derived for a given configuration of the system. Proposed reconfigurations can be evaluated through off-line tests using the formal model to determine whether the new mapping of operations-to-code blocks disrupts existing dataflow dependencies in the system. System administrators—or software components that control adaptivity in autonomic systems—can use the results of these tests to gauge the impact of a proposed reconfiguration on the existing system. The system model presented in this paper underpins the design of reconfigurable ARMOR (Adaptive Reconfigurable Mobile Objects of Reliability) processes that provide flexible error detection and recovery services to user applications.

Reconfigurability provides the foundation upon which autonomic systems can adapt to their changing environments, which is useful for dynamically optimizing system functionality based upon the observed execution profile or for recovering from errors and failures without human intervention. Unfortunately, reconfiguring distributed systems is difficult in practice, since the processes are often interdependent. Arbitrarily changing the behavior of one process through reconfiguration may render other processes unusable. By formally expressing the dependencies among processes, both static and dynamic reconfigurations can be analyzed to determine whether they are compatible with the existing configuration.

This paper describes a model for formally capturing the structure and run-time behavior of a distributed system. The structure is defined by a set of elements containing the state variables in the system. The run-time behavior is defined by threads that execute atomic actions called *operations*. Operations invoke code blocks to bring about state changes in the sys-

tem, but these state changes are limited to a single element and thread. The variables accessed during the execution of an operation constitute the signature of the operation. The collective set of operation signatures in the system can be used to derive dataflow dependencies between operations and between elements.

Reconfiguring the system is modeled by changing the code blocks that are invoked by an operation. Given the current configuration of the system, a proposed reconfiguration can be subjected to an off-line test to determine whether any of the dataflow dependencies of the system change as a result of the reconfiguration. If there are changes to the dataflow dependencies, the user is warned that the proposed reconfiguration may not be safe. Ultimately, the user decides whether or not to apply a reconfiguration.[1]

We have employed the proposed system model in developing a software-implemented fault-tolerant (SIFT) environment based around ARMORs (Adaptive Reconfigurable Mobile Objects of Reliability) to provide error detection and recovery for distributed applications. To ensure that the SIFT environment is resilient to failures, dedicated fault-tolerant mechanisms were added by reconfiguring the ARMOR processes[2] that make up the SIFT environment. The system model and associated criteria for safe reconfigurations were applied to verify that the additional fault-tolerant mechanisms did not impact the SIFT functionality in unexpected ways.

## System model

An abstract model that captures the structure and run-time execution of the system is used to analyze the effects of reconfiguration on the dependencies among the various components in the system. A system is specified by a triple $(\mathcal{C}, \mathcal{V}, \mathcal{T})$:

- $\mathcal{C}$ is the set of code blocks in the system. A code block performs a computation triggered by events called *operations*.
- $\mathcal{V}$ is the set of state variables[3] in the system. Variables are only accessed through executing code blocks.
- $\mathcal{T}$ is the set of threads in the system. Threads execute by sequentially invoking code blocks in the system through operations, which bring about state changes by manipulating the system variables in $\mathcal{V}$.

**Elements.** State variables in the system are partitioned into components called *elements*. Code blocks are placed in the elements containing the state variables manipulated by the code blocks. Given a predicate $Access(c, v)$ that is true if and only if code block $c \in \mathcal{C}$ reads or writes variable $v \in \mathcal{V}$, an element can be defined formally as follows.

Definition 1 (Element): *An element is a pair* $(C, V)$, *where V is a set of variables and C is a set of code blocks that do not access variables other than those in V:*

$$\text{element } e \equiv (C, V),$$
$$C \subseteq \mathcal{C} \wedge (\forall c \in C, V \subseteq \mathcal{V},$$
$$v \in \mathcal{V} : Access(c, v) \Rightarrow v \in V)$$

Elements partition the system so that each variable is found in one and only one element, and the code blocks within an element cannot access state variables residing in other elements.

**Operations.** Code blocks in the system are indirectly invoked through operations. One operation can invoke several code blocks as defined by a function *BindCode*: $\mathcal{P} \rightarrow 2^{\mathcal{C}}$, where $\mathcal{P}$ is the alphabet of operations and $2^{\mathcal{C}}$ is the power set over all code blocks (i.e., *BindCode* maps an operation to a set of code blocks). The notation "$p \rightarrow c$" is also used to denote that operation $p$ is bound to code block $c$ via the *BindCode* function. An operation executes by being delivered to all code blocks bound to the operation via the *BindCode* function.

Definition 2 (Operation Delivery): *An operation* $p \in \mathcal{P}$ *is said to be* delivered *to a code block* $c \in \mathcal{C}$, *denoted Deliver*$(p, c)$, *by executing code block c. The delivery completes when code block c completes executing.*

Because each code block exists in one and only one element, delivery can be thought of as occurring with respect to elements as well. An operation executes by being delivered to all bound code blocks (elements) in the system. Execution completes only after all deliveries for $p$ complete.

Definition 3 (Operation Execution): *An operation* $p \in \mathcal{P}$ *is said to have* executed, *denoted Execute*$(p)$, *after it has been delivered to all code blocks bound to p:*

$$Execute(p) \equiv \forall c \in BindCode(p) : Deliver(p, c)$$

**Threads.** The run-time behavior of the system is characterized by a set of threads, each of which se-

rially executes a sequence of operations. Given $\mathcal{P}$, the set of all operation sequences is given by $\mathcal{P}^*$. A sequence can be specified by its constituent operations as $P = \langle p_0, p_1, p_2, \ldots \rangle$. We denote $p$ to be in sequence $P$ by $p \in P$, and that $p$ precedes $q$ in sequence $P$ by $p \prec_P q$.

A thread $T \in \mathcal{T}$ is specified by a triple $T = (P, V, F)$:

- $P \in \mathcal{P}^*$ is a sequence of operations organized as a stack, also referred to as $T$.ops. Threads execute by sequentially executing the operations on the stack, beginning with the head operation. To emphasize that operations are executed within a thread, the notations used in Definitions 2 and 3 are extended to $T.Deliver(p, c)$ and $T.Execute(p)$.
- $V \in \mathcal{V}_{thd}$ is a subset of *private thread variables* that are only accessible through thread $T$. $\mathcal{V}_{thd}$ is disjoint from the state variables $\mathcal{V}$. Although two threads can contain private variables with the same names, each thread maintains its own independent value. The set of private variables for thread $T$ is also referred to as $T$.vars.
- $F$ is a data structure called a *frame stack*, which is used to preserve the contents of private thread variables across nested operation invocations, essentially providing scope to the variables in the thread. A nested operation invocation occurs when an operation pushes a sequence of operations onto $T$.ops.

The pseudocode for the execution of a thread is given in Figure 1. The *while* loop executes until the operation stack is exhausted. The head operation is popped from the stack and stored in variable $p$. Line 5 executes the operation. Note that because the statement $T.Execute(p)$ does not complete until all deliveries for $p$ complete (i.e., until all code blocks bound to $p$ have executed), the operations within a thread are executed in a strictly sequential order.

**Delivery actions.** The code blocks that execute during an operation delivery cannot make arbitrary state changes to the system. Their effects are confined to the current thread and the element to which the operation is delivered. Specifically, a single delivery of operation $p$ in thread $T$ to code block $c$ (i.e., $T.Deliver(p, c)$) can perform any of the following actions:

1. State variables of element $e$ (the element to which $c$ belongs) can be read. $c$.readVars denotes the set of state variables read by $c$.

Figure 1 Pseudocode for thread execution

```
1  procedure ExecuteThread(T) :
2      while T.ops≠∅ do
3          p := head(T.ops)
4          T.ops := tail(T.ops)
5          T.Execute(p)
6      end do
```

2. State variables of element $e$ can be written by code block $c$, denoted by $c$.writeVars.
3. Private variables within thread $T$ can be read by code block $c$, denoted by $c$.readThdVars.
4. Private variables within thread $T$ can be written by code block $c$, denoted by $c$.writeThdVars.
5. The code block can atomically push new operations onto the operation stack of $T$. Let $c$.pushOps refer to the *set* of operation sequences that can be pushed while $c$ executes. Only one of these sequences, however, is pushed when $c$ executes (allows the code block to push different sequences depending upon run-time conditions).
6. The code block can create new threads whose states are initialized from the private variables in the parent thread $T$ and the state variables in $e$.

These six items constitute a signature of code block $c$, and the signature provides a succinct, black-box description of how the code block manipulates element state and thread state during run time. Programmers specify the signature as meta-data for each code block that they develop. The collective set of signatures for all code blocks in the system are used to derive dataflow dependencies among operations given the current *BindCode* mapping function.

**Intrathread dependencies among operations.** Because elements are allowed to read from private thread variables during an operation delivery, a particular operation delivery may be dependent upon deliveries earlier in the execution of the thread that wrote to thread variables. To better understand the dependencies among operations within a thread, the concept of an input/output signature is developed with respect to individual operations and sequences of operations as follows:

- An *input signature for an operation* describes the thread variables that are read by an operation when it executes.

- An *output signature for an operation* describes the thread variables that are written when an operation executes.
- An *input signature for an operation sequence* describes the thread variables whose values must be established before a sequence begins executing. These variables serve as inputs to the aggregate sequence of operations.
- An *output signature for an operation sequence* describes the thread variables that are the intended outputs of a sequence of operations. An operation sequence is pushed onto the operation stack of a thread with the express purpose of writing to the thread variables in the output signature of the sequence.

***Input signatures.*** First, the thread variables used as input for operation delivery $T.Deliver(p, c)$ are considered. Define a function $DeliveryInputSig$: $\mathcal{P} \times \mathcal{C} \rightarrow 2^{\mathcal{V}_{thd}}$ as follows:

$$DeliveryInputSig(p, c) = c.\text{readThdVars} \cup \left( \left( \bigcup_{P \in c.\text{pushOps}} SeqInputSig(P) \right) - c.\text{writeThdVars} \right) \quad (1)$$

This definition states that the set of input variables used by $T.Deliver(p, c)$ not only includes $c$.readThdVars but also includes those thread variables used as input for the operation sequences pushed by code block $c$. The outermost parenthetical expression in Equation 1 includes the input variables of the sequence (denoted by $SeqInputSig(P)$, to be defined later), but excludes those variables used by the sequence that were written by $c$.[4] The intuition is that the input signature for $T.Deliver(p, c)$ should only include those variables whose values were defined *before* code block $c$ executed.

In general, operation $p$ can be delivered to several code blocks, and the composite input signature for operation $p$ denoted by $ExecInputSig(p)$: $\mathcal{P} \rightarrow 2^{\mathcal{V}_{thd}}$ can be derived from Equation 1:

$$ExecInputSig(p) = \bigcup_{c \in BindCode(p)} DeliveryInputSig(p, c)$$

The input signature for a sequence $P$ is slightly more complicated. If an operation $p \in P$ takes variable $v$ as input, then $v$ should not be part of the input signature for $P$ if $v$ was written by an operation that preceded $p$ in $P$. Define $SeqInputSig$: $\mathcal{P}^* \rightarrow 2^{\mathcal{V}_{thd}}$ as follows:

$$SeqInputSig(P) = \{v \in \mathcal{V}_{thd} : \exists\, p \in P : v \in ExecInputSig(p) \wedge \forall q \prec_P p : v \notin ExecOutputSig(q)\}$$

***Output signatures.*** As with input signatures, the output signature is defined first with respect to a single delivery $p \rightarrow c$ and then extended to account for all code blocks bound to $p$.

$$DeliveryOutputSig(p, c) = c.\text{writeThdVars} \cup \bigcup_{P \in c.\text{pushOps}} SeqOutputSig(P),$$

$$ExecOutputSig(p) = \bigcup_{c \in BindCode(p)} DeliveryOutputSig(p, c) \quad (2)$$

The delivery output signature in Equation 2 includes not only the variables written directly by $c$, but also the variables written by any sequence of operations pushed by $c$.

The output signature for an operation sequence $P$ is defined, in general, by the programmer. The programmer who writes code block $c$ pushes sequence $P$ with the intent of producing specific outputs. The individual operations in $P$ may write intermediate values as $P$ executes, but these intermediate results ultimately will be discarded when $P$ completes (see the discussion of the frame stack that follows). Discarding the intermediate values produced by $P$ provides scope to the thread variables, and this is similar to saving or restoring register contents when entering or exiting a function in order to preserve the contents of registers not intended to store function outputs. Because the intended output of an operation sequence cannot be derived without semantic knowledge of the sequence, the output signature $SeqOutputSig(P)$ must be specified by the programmer.

***Frame stack.*** Each thread contains a frame stack that implements the saving or restoring of private thread variables described above. When an operation sequence is pushed on the operation stack of the thread, the current values of all intermediate output variables for the sequence are saved. This set consists of all thread variables that will be written by the operations in the pushed sequence $P$, exclud-

ing those variables designated as the intended output of the sequence:

$$\bigcup_{p \in P} ExecOutputSig(p) - SeqOutputSig(P)$$

The variables stored in the top entry of the frame stack are restored when the operation sequence $P$ completes.

**Concurrency.** Operations within a thread execute in a sequential order. Only one thread executes within an element (i.e., delivers operations to code blocks within an element) at any given time to ensure mutually exclusive access to the state variables of the element. This assumption treats each delivery as if it were writing to the element, but this requirement can be loosened if the deliveries can be tagged as "read-only" or "read/write," in which case a multiple-read, single-writer lock can be used to control access to the elements.

**Applying the model to object-oriented programs.** In object-oriented systems, objects are similar to elements in the sense that objects typically encapsulate state and member functions that operate on the state of the object. Each object, therefore, has an associated signature for each of its member functions. When these member functions are invoked within a thread, dataflow dependencies exist among the objects visited by the execution of a thread. This is the intuition behind the construction of the model presented in this section.

Elements, however, more generally represent sets of related objects. For example, a tree data structure may represent each node as an object, but the entire tree would most likely be encapsulated in an element. Keeping I/O signatures at the coarser granularity level of elements makes the dataflow dependability analysis more manageable than if per-object signatures were required.

Member function invocations, therefore, are modeled as operations being delivered to elements. Nested function invocations are split into multiple operation deliveries, since the model defines the execution of a thread as the sequential execution of operations. For example, consider a function $X$ that performs some computation $C_1$, calls function $Y$, and performs an additional computation $C_2$ after $Y$ returns. This execution trace is modeled as three operation deliveries: the execution of $C_1$, followed by the execution of $Y$, followed by the execution of $C_2$. Each of these operation deliveries represents an atomic state change to the system state.[5]

With programs explicitly designed around the system model presented in this paper, a structure is in place for extracting information to perform dataflow analysis on the individual operations that make up the execution of the system. Furthermore, the model allows the information to be gathered and analyzed in an automated fashion, given the I/O signatures of each individual operation. If the dataflow analysis framework is to be applied to object-oriented programs not designed with the proposed system model in mind, then the issue becomes one of how to extract the appropriate information needed for the dataflow analysis (e.g., how do objects pass information between each other within a thread of execution, what information is exchanged through each object invocation, and what are the effects of each object's invocation on system state). After gathering this information, the effects of a reconfiguration on the system can be determined. Obtaining such information may require manual input from the programmer or may require refactoring certain aspects of the application to more closely conform to a system model such as the one proposed in this paper to facilitate the automated dataflow analysis of proposed reconfigurations.

## Reconfigurability

The sets of code blocks, elements, and operations are considered to be static, and reconfiguration occurs by either adding or removing a single operation binding (i.e., changing the *BindCode* function). The system transitions into a new configuration view whenever the *BindCode* function changes. A configuration view is denoted by $\mathbb{V}_i$, with configuration view $\mathbb{V}_{i+1}$ occurring immediately after $\mathbb{V}_i$. When necessary, the configuration view index will be specified along with the *BindCode* function (e.g., $BindCode_i(p)$ specifies the set of code blocks bound to $p$ under configuration view $\mathbb{V}_i$).

**Impact on thread state.** The *BindCode* function determines the I/O signatures for operations and, therefore, establishes dependencies among operations within a thread of execution. Programmers design code blocks with an understanding of these dependencies in order to bring about a desired effect. For example, if operation $p_k$ copies data from element $e_1$ to variable $v$, and operation $p_{k+1}$ copies data from variable $v$ to element $e_2$, data are transferred from one $e_1$ to $e_2$ by virtue of the I/O signatures of $p_k$ and

$p_{k+1}$. If a reconfiguration—either adding or removing an operation binding—disrupts the data dependencies for future operations in the execution of a thread, then the reconfiguration is said to be unsafe.

Definition 4 (Safe Reconfiguration): *A reconfiguration from view* $\mathbb{V}_i$ *to* $\mathbb{V}_{i+1}$ *that affects the binding* $p \rightarrow c$, *where* $p \in \mathcal{P}$ *and* $c \in \mathcal{C}$, *is considered to be* safe with respect to thread state *if and only if, for all executed operation sequences* $P \in \mathcal{P}^*$ *in which* $p \in P$:

1. *All dataflow dependencies*[6] *between operations following* $p$ *and operation* $p$ *in view* $\mathbb{V}_i$ *continue to exist in view* $\mathbb{V}_{i+1}$.

2. *All dataflow dependencies between operations following* $p$ *and operation preceding* $p$ *in view* $\mathbb{V}_i$ *continue to exist in view* $\mathbb{V}_{i+1}$:

$$\forall P \in \mathcal{P}^* : p \in P \Rightarrow \forall q \preceq_P p, r \succ_P p:$$

$$r \text{ dataflow-dependent on } q \text{ in } \mathbb{V}_i \Rightarrow r \text{ dataflow-dependent on } q \text{ in } \mathbb{V}_{i+1}$$

System reconfigurations change the binding of operation $p$, either assigning a code block to $p$ (causing an extra computation to be performed) or removing a code block from $p$ (eliminating a computation from threads that execute in the system). A reconfiguration is unsafe only if it affects operations further downstream in the execution of the thread. The following examples illustrate safe reconfigurations in which dataflow dependencies are added, removed, or changed:

- A new binding $p \rightarrow c$ reads from some thread variable $v$, resulting in a new dataflow dependency that did not exist in the previous configuration. This reconfiguration is safe, since it does not disrupt the execution of operations further downstream.

- A removed binding $p \rightarrow c$ causes $p$ to no longer read from thread variable $v$. Obviously, this reconfiguration destroys a dataflow dependency that previously existed, but the reconfiguration is not considered to be unsafe since this, in itself, does not affect operations further downstream in the execution of the thread.

  After the binding $p \rightarrow c$ is removed, it may be safe to remove the binding that wrote to $v$ earlier in the execution of the thread. This is an example of how functionality often can be removed incrementally: the safe reconfiguration criteria help identify the correct order in which multiple bindings should be removed to bring the system safely to a desired configuration.

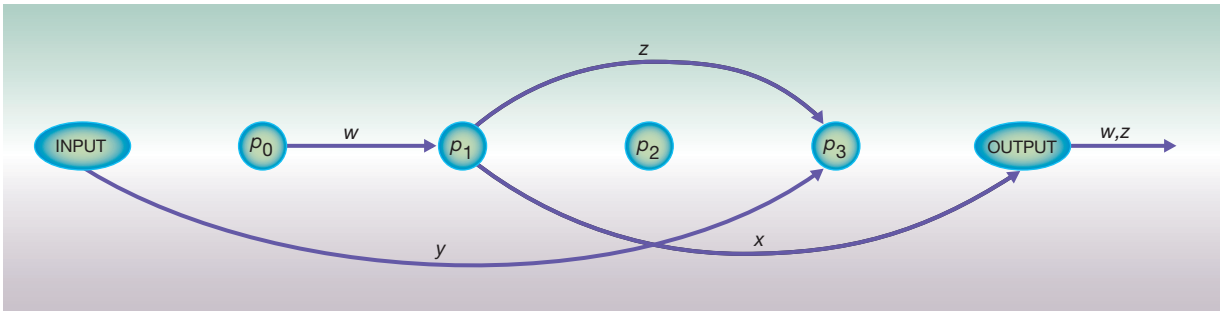
- A new binding $p \rightarrow c$ writes to thread variable $v$ that is not currently read by any other operation further downstream in the execution of the thread. Although this lone reconfiguration does not immediately bring about a change in functionality, additional reconfigurations can be made so that operations further downstream read from thread variable $v$. As an alternative, a reconfiguration can be made so that a later operation pushes a new operation sequence onto the operation stack of the thread with the express purpose of performing some computation based upon the newly written value in variable $v$.

As the previous examples suggest, several reconfigurations can be made to remove or replace specific functionality in the system. The dataflow dependency analysis assists in deciding the proper order to apply such reconfigurations so that the correct behavior of the system is not disturbed. It should be clear, however, that the safe reconfiguration criteria presented in this section cannot guarantee consistency using dataflow dependency analysis alone. Definition 4 is a sufficient but not necessary condition for preserving consistency across reconfigurations. A semantic analysis of the system is required to make this consistency determination, and the semantics of a code block cannot be derived solely from the code block I/O signatures. Dataflow analysis, however, permits the programmer to understand the relationships between a new code block and the other code blocks in the system, simply by incorporating the signature of the new code block into the following analysis framework.

**Dataflow analysis framework.** The analysis framework and resulting criteria depend only upon the current system configuration (i.e., the $BindCode_i$ function and resulting I/O signatures) and the proposed configuration (i.e., $BindCode_{i+1}$). If the criteria indicate that the proposed reconfiguration may be unsafe, the user is notified of the existing dataflow dependency that would be broken by the reconfiguration. Other reconfigurations may be needed to compensate for the broken data dependency.

Figure 2, used throughout this section as a running example, shows data dependencies among operations in a sequence $P = \langle p_0, p_1, p_2, p_3 \rangle$ that was

Figure 2 Read/write dependencies among operations in sequence $p = \langle p_0, p_1 p_2, p_3 \rangle$




pushed onto the operation stack. The arcs between nodes indicate the dataflow dependencies between operations (e.g., operation $p_1$ writes to a variable $z \in \mathcal{V}_{thd}$, which is later read by $p_3$). Arcs emanating from input define the input signature for $P$ (e.g., $p_3$ reads from variable $y$, whose value was not altered by either $p_1$ or $p_2$). Arcs that point to the output node represent the variables that are part of the output signature for sequence $P$. Variables that store intermediate values are shown to emanate from the output node, indicating that they will be restored when $P$ completes. A dataflow dependency graph can be constructed completely for any sequence $P$ given the I/O signatures for $P$ and the individual operations in $P$.

The dataflow dependencies in Figure 2 can be altered by either adding or removing an operation binding. Since the analysis performed in both cases is similar, the discussion focuses on adding an operation binding $p \rightarrow c$, without loss of generality. Adding a binding $p \rightarrow c$ is considered to be safe if $DeliveryOutputSig(p, c) = \emptyset$, since thread variables are not written when $p$ is delivered to $c$.

If $DeliveryOutputSig(p, c) \neq \emptyset$, then how the additional writes affect later operations in the thread must be examined. For illustrative purposes, suppose that the binding $p_2 \rightarrow c$ is being added to the example in Figure 2. The analysis decomposes the effect of the additional write into two cases:

1. How the additional write from $p_2 \rightarrow c$ affects later operations within $P$ (intrasequence dependencies)
2. How the additional write from $p_2 \rightarrow c$ affects operations that follow execution of $P$ in the thread (extrasequence dependencies)

***Intrasequence dependencies.*** The first case examines how adding binding $p \rightarrow c$ affects the dataflow among operations within the sequence to which $p$ belongs. If an operation $p'$ that follows $p$ in sequence $P$ reads from a variable written by the delivery $T.Deliver(p, c)$, then $p'$ is dependent upon the reconfiguration if no other operation between $p$ and $p'$ overwrote variable $v$. This condition is formally expressed in the following theorem (proofs are omitted for the sake of brevity).

Theorem 1 (Intrasequence Safety): *Given an operation sequence $P \in \mathcal{P}^*$ and an operation $p \in P$, the binding $p \rightarrow c$ can be safely added to the system only if any variables written by $p \rightarrow c$ do not overwrite the values in the variables expected by later operations within P. Formally, a safe configuration implies:*

$$\forall p' \succ_P p : V = (ExecInputSig(p') \cap DeliveryOutputSig(p, c)) \wedge V \neq \emptyset \Rightarrow \forall v \in V: \exists p'' \in P : p \prec_P p'' \prec_P p' \wedge v \in ExecOutputSig(p'').$$

In the following examples, the binding $p_2 \rightarrow c$ is added to the system described in Figure 2. The four cases differ with respect to the variables written by the new binding (i.e., $DeliveryOutputSig(p_2, c)$). The preceding theorem is applied to each case to determine whether the proposed reconfiguration is considered to be safe:

1. Write to $z$. This reconfiguration is not safe, since it changes the dataflow dependency graph in Figure 2: $p_3$ would read the value written to $z$ by $p_2$, not the value written by $p_1$ as before (Figure 3).
2. Write to $y$. This reconfiguration is also not safe

Figure 3 Write to *z* reconfiguration

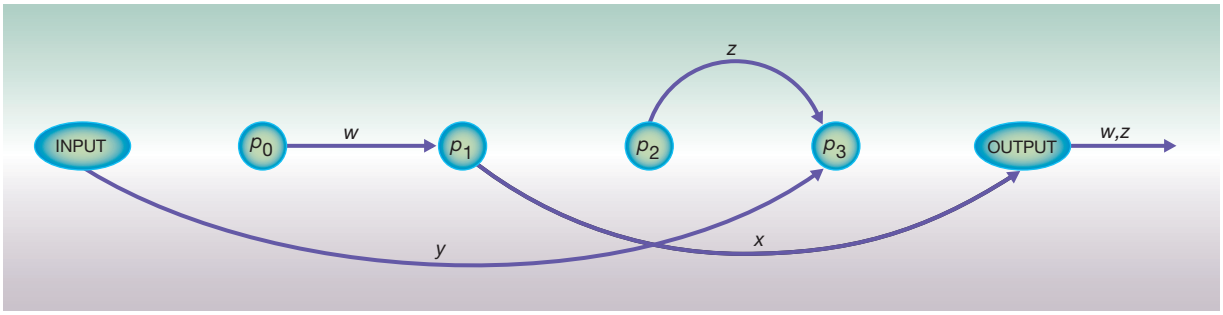
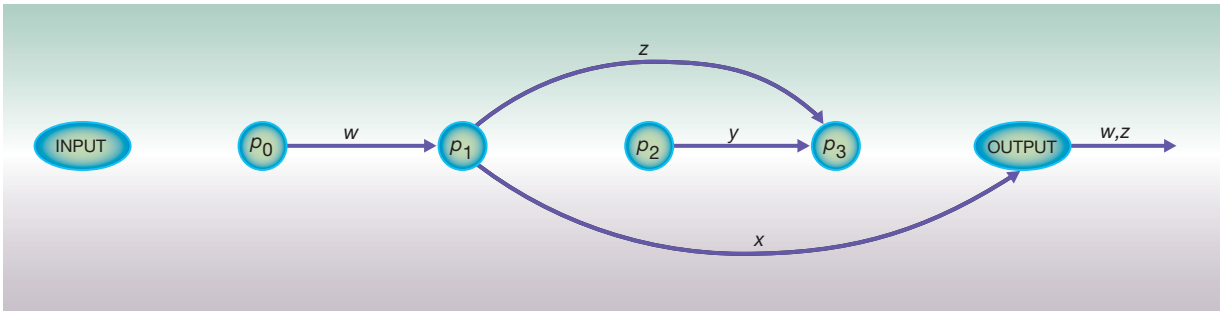



Figure 4 Write to *y* reconfiguration




for similar reasons: $p_3$ no longer reads the value of $y$ established before $P$ executes (Figure 4).

3. Write to $w$. This reconfiguration is safe, since no operation within $P$ after $p_2$ reads from $w$. Since $w$ is an intermediate variable (not the final output of $P$), its value will be restored when $P$ completes; thus, the side effect of the new binding $p_2 \rightarrow c$ will be masked (Figure 5).
4. Write to $v$. This reconfiguration is also safe, since no operation within $P$ after $p_2$ reads from $v$ (Figure 6).

The cases involving writing $w$ and writing $v$ differ in one important respect: in the latter case, variable $v$ was not included in the set of intermediate thread variables used by $P$, since no operation in $P$ was known to produce $v$ prior to the reconfiguration. Thus, there are two possible scenarios to consider:

- Sequence $P$ was pushed onto the operation stack of the thread prior to the reconfiguration, in which case the frame stack of the thread does not contain the correct value of $v$ to restore when $P$ completes. Because sequence $P$ was pushed onto the stack with the intent of being executed in an earlier configuration not containing the binding $p_2 \rightarrow c$, it is appropriate to suppress the $p_2 \rightarrow c$ delivery, thus preventing variable $v$ from being overwritten as a side effect of the reconfiguration.
- Sequence $P$ was pushed onto the operation stack of the thread after the reconfiguration, in which case $v$ is known to be an intermediate value produced by $P$. The stack frame, therefore, contains the correct value of $v$ to restore when $P$ completes.

***Extrasequence dependencies.*** The additional binding $p_2 \rightarrow c$ from Figure 2 can affect only operations that follow sequence $P$ in the execution of the thread if $p_2 \rightarrow c$ writes to variables in the output signature of $P$. This is a necessary but not sufficient condition for the reconfiguration to be considered safe, since $p_2 \rightarrow c$ can write to an output variable of $P$ as long as no other operation that follows $P$ depends upon the overwritten value (again, the notion of preserving the dataflow dependency relationships that existed before reconfiguration).

Figure 5 Write to *w* reconfiguration

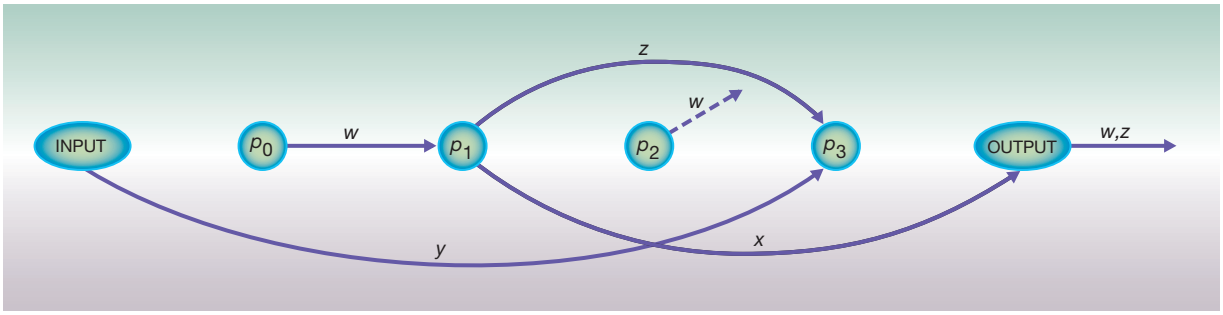



Figure 6 Write to *v* reconfiguration

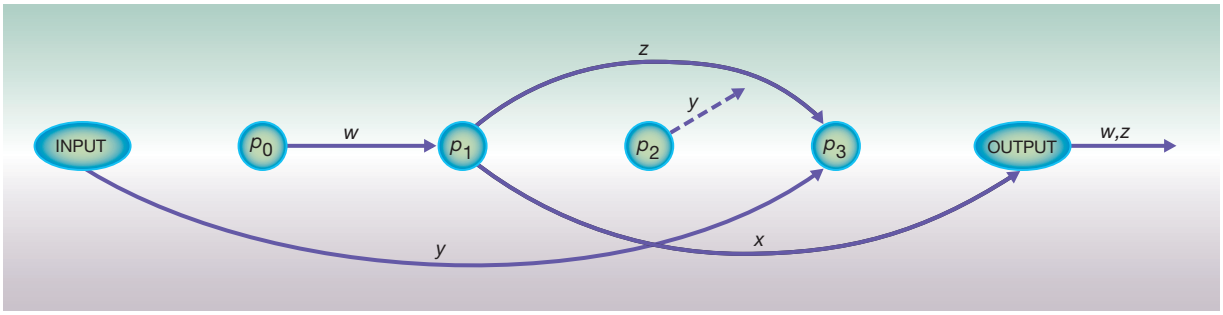



Figure 7 presents an example in which operation $p$ executes as part of a sequence $P$ pushed by an operation $q_j \in Q$. When sequence $P$ completes, operations that follow $q_j$ in sequence $Q$ will begin executing ($q_{j+1}$ in the figure). If $p$ is the last operation within $P$ to write to thread variable $v$, and $v$ is in the output signature for sequence $P$, then the value written to $v$ by operation $p$ will propagate to operations that follow $q$ in sequence $Q$.

Lemma 1 (Write Propagation Outside of Sequence): *Given an operation sequence P and an operation* $p \in P$, *the value written to variable* $v$ *by* $p$ *will propagate to operations that follow P if and only if no other operations after* $p$ *in P overwrite variable* $v$. *This condition is defined by the following predicate:*

$$WritePropagation(v, p, P) \equiv v \in (ExecOutputSig(p) \cap SeqOutputSig(P)) \wedge \forall p' >_P p : v \notin ExecOutputSig(p')$$

If operation $q \in Q$ pushes operation sequence $P$ onto the operation stack of the thread, then $q$ is called the *parent operation* of sequence $P$. In general, an operation sequence can have several parents, given by the set $parents(P)$ as follows:

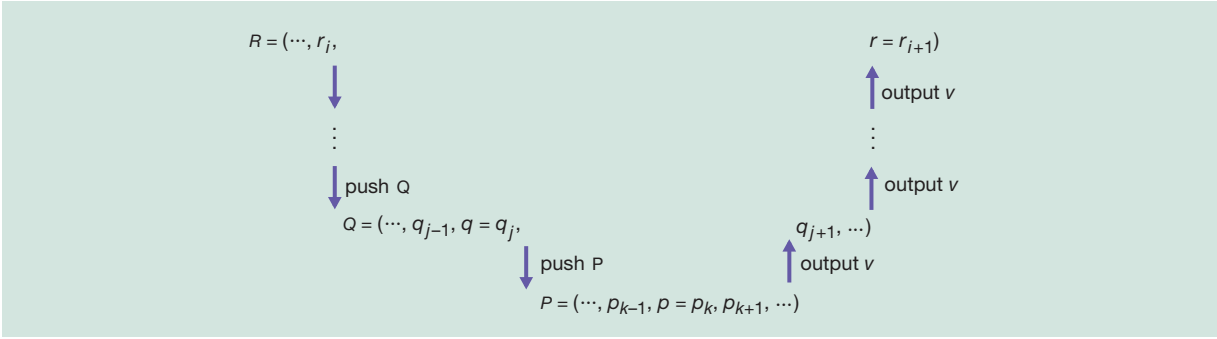
$$parents(P) \equiv \{p \in \mathcal{P} : \exists c \in \mathcal{C} : P \in c.\text{pushOps} \wedge c = BindCode(p)\}.$$

The concept of parent operations can be used to establish a "leads to" relationship between two operations and between an operation and a sequence:

- $q \rightsquigarrow P \Leftrightarrow q \in parents(P)$
- $q \rightsquigarrow p \Leftrightarrow p \in P \wedge q \in parents(P)$

Finally, $p \overset{*}{\rightsquigarrow} q$ denotes a transitive chain of "leads to" relationships, meaning that the execution of operation $p$ brings about the execution of operation $q$ in either one step (i.e., $p \rightsquigarrow q \Rightarrow p \overset{*}{\rightsquigarrow} q$) or several steps (i.e., $p \overset{*}{\rightsquigarrow} q \wedge q \overset{*}{\rightsquigarrow} r \Rightarrow p \overset{*}{\rightsquigarrow} r$). The "leads

Figure 7 Output variable $v$ propagates after being written by operation $p$




to" relationship is needed to determine the extent to which a reconfiguration affects other operations. If $q \overset{*}{\rightsquigarrow} p$, then changes to the binding of $p$ can propagate to operations that follow $q$ as expressed in Lemma 2, presented below. The dataflow dependencies for each operation that leads to $p$ must be checked in order to determine whether a reconfiguration is safe.

Continuing with the example in Figure 3, let operation $r \in R$ push sequence $Q$ onto the operation stack of the thread when $r$ executes; thus, $r \overset{*}{\rightsquigarrow} p$. The value in variable $v$ that exists after operation $q$ executes ($q$ being the operation that pushed sequence $P$) will propagate to operations that follow $r$ in sequence $R$ if and only if the remaining operations in $Q$ do not overwrite $v$ and $v$ is part of the output signature of sequence $Q$. This propagation continues up the $\overset{*}{\rightsquigarrow}$ chain as long as these two conditions hold at every step; as soon as one of the conditions is violated, checking can stop, since the value in $v$ will not propagate further.[7]

Lemma 2 (Generalized Write Propagation): *Given an operation sequence $P$ for which $r \overset{*}{\rightsquigarrow} P$, the value written to variable $v$ by operation $p \in P$ propagates to operations that follow $r$ if and only if the following condition holds:*

$$WritePropagation^*(v, p, P, r)$$
$$\equiv \begin{cases} WritePropagation(v, p, P), & for\ r \rightsquigarrow p \\ \exists\, q \in parents(P) : r \overset{*}{\rightsquigarrow} q \rightsquigarrow p \wedge q \in Q \Rightarrow \\ \quad WritePropagation(v, p, P) \wedge & otherwise \\ \quad WritePropagation^*(v, q, Q, r), \end{cases}$$

Proof rationale. The *WritePropagation** predicate is intended to be a generalization of the *WritePropagation* predicate introduced in Lemma 1. When $r$ is an immediate parent of $p$ (i.e., $r \rightsquigarrow p$), Lemma 1 can be directly applied.

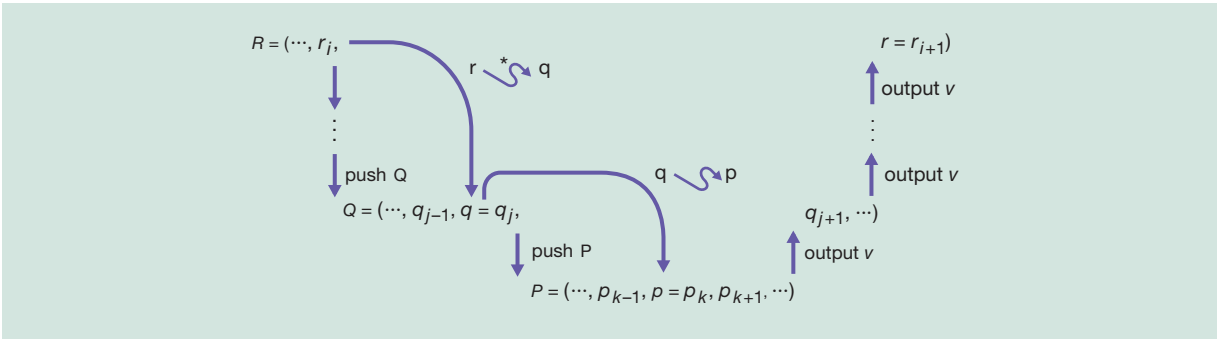
If $r \overset{*}{\rightsquigarrow} p$ forms a multistep chain (see Figure 8), then *WritePropagation** is recursively defined. The intuition is that the write to variable $v$ performed by operation $p$ will propagate back to operation $r$ if there is at least one path of propagation, namely a path through operation $q$ in which $q$ is the parent operation of $p$. The write propagates from $p$ to $r$ if the write propagates outside sequence $P$, denoted by $WritePropagation(v, p, P)$, and if the write propagates from $q$ to $r$, denoted by $WritePropagation^*(v, q, Q, r)$.

If the value in $v$ propagates back to sequence $R$ in Figure 7, then a reconfiguration that overwrites $v$ can be considered safe as long as future operations in $R$ do not read the overwritten value in $v$. This is the essence of the following theorem: a reconfiguration is considered safe only if either (1) the effects of the new binding do not propagate to sequence $R$ (where $r \overset{*}{\rightsquigarrow} p$ and $r \in R$) or (2) the effects of the new binding propagate to sequence $R$, but no operations within $R$ read from the overwritten variable.

Theorem 2 (Extrasequence Safety): *Given an operation sequence $P \in \mathcal{P}^*$ and an operation $p \in P$, the binding $p \rightarrow c$ can be safely added to the system only if any variables written by $p \rightarrow c$ do not overwrite the values in the variables expected by later operations that follow $P$. Formally, a safe reconfiguration implies:*

$$\forall R \in \bigcup_{c \in \mathcal{C}} c.\text{pushOps} : \forall r \in R : r \overset{*}{\rightsquigarrow} p$$
$$\Rightarrow (\forall v \in \mathcal{V} : \neg WritePropagation^*(v, p, P, r)$$

Figure 8 Value written by operation $p$ propagates back to sequence $R$ through intermediate sequence $Q$




$$\vee\ (\forall r' \succ_R r\ :\ v \in ExecInputSig(r')$$
$$\Rightarrow \exists\ r'' \in R\ :\ r \prec_R r'' \prec_R r'$$
$$\wedge\ v \in ExecOutputSig(r'')))$$

Proof rationale. All operations that lead to $p$ must be examined to determine whether the reconfiguration is safe. Only those operations $r$ that exist in operation sequences pushed by code blocks are considered ($r \in R$, where $R \in \cup_{c \in \mathcal{C}} c.\text{pushOps}$). For each of these operations, one of the following two conditions must hold for each thread variable for the reconfiguration to be considered safe:

1. The value written by $p$ from sequence $P$ must not propagate back to those operations that follow $r$ in sequence $R$ ($\neg WritePropagation^*(v, p, P, r)$). If the value does not propagate, then operations that follow $r$ will not be affected by the reconfiguration.
2. If the value propagates to sequence $R$, then operations that follow $r$ in $R$ are checked to see whether they read from the variable written by $p$ ($v \in ExecInputSig(r')$, where $r' \succ_R r$). If such an operation $r'$ is found, then the reconfiguration can be considered safe only if there is another operation between $r$ and $r'$ that overwrites the value in $v$ (i.e., $r'$ does not read the value in $v$ established by operation $p$).

Theorem 3 (Safe Reconfiguration Criteria): *Adding a new binding $p \rightarrow c$ to the system is considered safe if and only if the new binding satisfies both the Intrasequence Safety and Extrasequence Safety properties.*

**Impact on element state.** The previous subsection established the conditions under which a reconfiguration can be considered safe from the standpoint of thread state. In addition to thread state, the system also contains state variables found in the elements. Operations in the threads can manipulate state variables as they execute, so changing the bindings of operations to elements can impact the changes that are brought about in the element state.

Recall from the subsection on concurrency that threads lock an element before an operation delivery to ensure mutually exclusive access to the element. The element is unlocked after the delivery, permitting other threads to operate on the element. For example, the execution of an operation sequence $P = \langle p_1, p_2, p_3 \rangle$ with operation-to-element bindings[8] $p_1 \rightarrow e_1$, $p_2 \rightarrow e_2$, and $p_3 \rightarrow e_1$ is shown in Figure 9A. State changes to the elements are atomic only with respect to a single operation delivery. Another thread, for example, can deliver an operation to element $e_1$ in Figure 9A between operations $p_1$ and $p_3$.

Figure 9B shows how multidelivery locks can be used to extend atomicity across several operation deliveries. In this case, the lock action indicates that the lock for each element is not released after delivery until the unlock is reached, at which point all the element locks in the block are released.[9,10] With multidelivery locks, dataflow dependencies among elements can be established within each lock/unlock block. Outside these blocks, the element dataflow dependencies cannot be determined, since other threads can overwrite the element state between operation deliveries. Once the dataflow dependencies are established within the lock/unlock blocks, the analysis for determining a safe reconfiguration is similar to the procedure outlined earlier in the subsec-

Figure 9 (A) Default locking for each operation delivery; (B) element locks held across several deliveries

(A) [**lock** $e_1$], $p_1$, [**unlock** $e_1$], [**lock** $e_2$], $p_2$, [**unlock** $e_2$], [**lock** $e_1$], $p_3$, [**unlock** $e_1$]

(B) [**lock**], $p_1$, $p_2$, $p_3$, [**unlock**]

tion, "Impact on thread state," but space considerations preclude a detailed examination of the criteria.

## Reconfigurations in practice

The safe reconfiguration criteria presented in the previous section require only static information for input (namely, the current and proposed configurations expressed as *BindCode* functions). Thus, these checks can be performed off line while the system executes. This section briefly describes examples in which reconfiguration is useful and how the safe reconfiguration criteria can be employed. It then describes a reconfigurable software-implemented fault-tolerant environment developed using these ideas.

**Example applications of reconfigurability.** The following are three types of applications in which the model is useful in determining the safety of a proposed reconfiguration:

1. Different execution phases for long-running applications. Some long-running applications require functionality that varies according to their phase of execution. A spacecraft sent to explore one of the outer planets in the solar system, for example, spends most of its time traveling to reach the target planet. While in this cruise mode, power must be conserved, and the demands on the system are few. When the spacecraft reaches the planet, however, it must perform several tasks, such as collecting data, taking pictures, navigating a fly-by of the planet, or controlling its own descent and landing. These phase-specific code blocks can be activated only when necessary through the reconfiguration concepts outlined in this paper. Since the number of phases and the transitions between phases is known at design time, the systems engineer can apply the safe reconfiguration criteria during the development cycle to verify that the intended transitions are safe.
2. On-line software upgrades. Since software upgrades involve changing the code that applications execute, on-line software upgrades can be viewed as reconfigurations to the system. Before the upgrades are made, the safe reconfiguration criteria can be used to ensure that the proposed upgrade is compatible with the existing configuration of the software.
3. Adaptivity. Some application domains require that the software adapt to changing conditions in the environment of the application. Middleware that provides fault tolerance to distributed applications, for example, may need to adjust the level of service it provides to the application depending upon the observed error behavior. Software structured around the system model presented in this paper facilitates this adaptation.

   Reconfiguration can be used to transform the computation of the application to take advantage of various mechanisms of fault tolerance (e.g., incremental checkpointing of element state,[11] alternate implementations of an element that employ design diversity to mitigate the effects of software bugs, and backup elements that store redundant copies of the data). The safe reconfiguration criteria can be used to show that the additional fault tolerance mechanisms do not disrupt the normal computation performed by the application.

**Reconfigurable SIFT environment.** We have developed a software-implemented fault-tolerant (SIFT) environment by employing the proposed formal system model. The SIFT environment consists of ARMOR processes, which provide error detection and recovery services to themselves and to user applications.[2,12] Because ARMOR processes are designed around the system model presented in this paper, the SIFT environment can be customized—even during run

time—to the particular dependability needs of the application.

The ARMOR-based SIFT environment was used for managing parallel scientific applications executing on a computing test bed at the Jet Propulsion Laboratory.[13] Extensive fault injection testing revealed that it is essential for the SIFT environment to be protected against errors in order to provide adequate fault-tolerance services to the application. The reconfigurability concepts introduced in this paper were applied to incorporate fault tolerance into the ARMOR processes as follows:

1. Microcheckpointing[11] was transparently added to the ARMOR processes to protect the state of the SIFT environment. The partitioning of the system state into elements permitted incremental checkpoints to be taken on an element-by-element basis. The microcheckpointing algorithm exploited the fact that state changes brought about by an operation delivery were confined to a single element and thread, thus making transparent checkpointing possible.
2. Assertion checks were added to strengthen error detection. These assertion checks were inserted during run time by dynamically changing the bindings of selected operations to pass through the assertion before being delivered to the original element. This construct was particularly useful in implementing range or sanity checks on inputs.

The safe reconfiguration criteria were applied in both of these cases to show that the added fault tolerance mechanisms did not disturb the existing functionality of the SIFT environment, which included running the scientific applications, recovering from application failures, recovering from node failures, and monitoring resource usage.

## Conclusion and related work

This paper has presented a model that captures the structure (defined by elements) and run-time behavior (defined by operations) of a system. An executing code block can bring about state changes only within a single element and thread. The extent of these state changes are represented in a signature for the code block, and the collective set of signatures in the system can be used to analyze the dependencies that exist among operations and among elements.

By indirectly invoking code blocks through operations, the behavior of the system can be reconfigured by changing the operation-to-code block binding. Indirection can be achieved statically through designs such as the Polylith software bus,[14] in which components are interconnected through a mediator. The bindings of operations to code blocks is somewhat similar to a publish-and-subscribe event subsystem, except that operations are not asynchronous. Operations execute sequentially and, therefore, more closely resemble instructions that execute in a virtual machine architecture described by the proposed system model.

Architectural description languages (ADLs) are popular in describing the structure of a system.[15–17] ADLs model the system at a conceptual level, using port-based connections between components to define the system structure. Some incorporate semantics that describe the behavior of the components and connectors, including constraints in their usage. Our model, in contrast, is rooted in the implementation of the system and is a bottom-up approach in which the programmer describes the behavior of code blocks through the use of per-block signatures. These signatures are processed in an automated fashion to construct dependency relationships between the system components. Additional code blocks can be designed without the need to formally incorporate them into a larger architectural description model—only the signatures for the new code blocks are required to apply the safe reconfiguration criteria presented earlier.

Nevertheless, several ADLs express dynamic reconfigurations at the architectural level. Darwin, for example, addresses the problem from a structural perspective by allowing components to be instantiated during run time,[18] but reconfigurations only occur while the system is quiescent to preserve consistency. Wright permits the architectural topology of the system to change during run time in response to special control events, which are distinguished from the usual communication events that drive its component behavior.[19] It has also been suggested that architectural styles and models can be incorporated into the adaptation framework of self-repairing systems as first-class entities.[20] The architectural styles are used to determine what aspects of the system should be monitored and how to reconfigure the system within predetermined constraints.

Shrivastava and Wheater describe a reconfigurable workflow model that provides run-time support for interconnecting the I/O of executing tasks.[21] Although the work outlines the infrastructure that supports re-

configuration, there is no mention of how to judge whether a proposed reconfiguration can be considered safe given the current configuration of the system. Reconfiguration compatibility issues have been addressed with respect to real-time control applications in Feiler and Li,[22] but this analysis requires knowledge of the semantics and permissible behavior of the application.

## Acknowledgments

This work was supported in part by NASA Fellowship NGT5-50228 for Keith Whisnant, NSF grants CCR 00-86096 ITR and CCR 99-02026, grant ARMY WMH 0993, and a grant from Motorola. Special thanks to Fran Baker for reviewing early drafts of this paper.

## Cited references and notes

1. It may be that the user is reconfiguring the system with the explicit intent of changing the dataflow dependencies; thus, the off-line tests are used only as warnings of possible incompatibility.
2. Z. Kalbarczyk, R. K. Iyer, S. Bagchi, and K. Whisnant, "Chameleon: A Software Infrastructure for Adaptive Fault Tolerance," *IEEE Transactions on Parallel and Distributed Systems* **10**, No. 6, 560–579 (June 1999).
3. Where appropriate, the *name* of a variable $v \in \mathcal{V}$ ($v$.name) is distinguished from the *value* of the variable ($v$.val).
4. If a code block both writes to thread variables and pushes an operation sequence onto the operation stack of the thread while operation $p$ is delivered, then the operations in the pushed sequence begin executing only after $p$ completely executes. The thread variables written by $p$, therefore, are visible to the operation sequence pushed by $p$.
5. When the boundaries of an atomic state change must extend beyond a single operation delivery, multidelivery locks can be used as described in a later section.
6. Dataflow dependencies can be formally defined between an operation $q$ and an earlier operation $p$ with respect to a particular thread variable $v$.
7. This situation can be compared to a series of nested function calls: The return value for the innermost function call only propagates to the topmost level if it is returned unmodified by each function call in the chain.
8. Since each code block exists in one and only one element, operation-to-element bindings can be derived from the *BindCode* function.
9. Details of these multidelivery locks (such as deadlock detection and avoidance[10]) are beyond the scope of this paper, but the issues are similar to those found in traditional concurrency control and parallel programming.
10. C.-S. Shih and J. Stankovic, *Survey of Deadlock Detection in Distributed Concurrent Programming Environments and Its Application to Real-Time Systems and Ada*, Technical Report UM-CS-1990-069, University of Massachusetts, Amherst, MA (August 1990).
11. K. Whisnant, Z. Kalbarczyk, and R. K. Iyer, "Micro-checkpointing: Checkpointing for Multithreaded Applications," *Proceedings of the 6th IEEE International On-Line Testing Workshop* (July 2000).
12. S. Bagchi, B. Srinivasan, K. Whisnant, Z. Kalbarczyk, and R. Iyer, "Hierarchical Error Detection in a SIFT Environment," *IEEE Transactions on Knowledge and Data Engineering* **12**, No. 2, 203–224 (March/April 2000).
13. K. Whisnant, R. K. Iyer, P. Jones, R. Some, and D. Rennels, "An Experimental Evaluation of the REE SIFT Environment for Spaceborne Applications," *Proceedings of the 2002 International Dependable Systems and Networks* (June 2002), pp. 585–595.
14. J. Purtilo, "The Polylith Software Bus," *ACM Transactions on Programming Languages and Systems* **16**, No. 1, 151–174 (January 1994).
15. D. C. Luckham, J. J. Kenney, L. M. Augustin, J. Vera, D. Bryan, and W. Mann, "Specification and Analysis of System Architecture Using Rapide," *IEEE Transactions on Software Engineering* **21**, No. 4, 336–355 (April 1995).
16. N. Medvidovic and R. Taylor, "A Classification and Comparison Framework for Software Architecture Description Languages," *IEEE Transactions on Software Engineering* **26**, No. 1, 70–93 (January 2000).
17. M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, and G. Zelesnik, "Abstractions for Software Architecture and Tools to Support Them," *IEEE Transactions on Software Engineering* **21**, No. 4, 314–335 (April 1995).
18. J. Kramer and J. Magee, "Analysing Dynamic Change in Software Architectures: A Case Study," *Proceedings of the Fourth International Conference on Configurable Distributed Systems* (1998), pp. 91–100.
19. R. Allen, R. Douence, and D. Garlan, "Specifying and Analyzing Dynamic Software Architectures," *Proceedings of the Conference on Fundamental Approaches to Software Engineering*, *Lecture Notes in Computer Science*, Vol. 1382 (April 1998).
20. S.-W. Cheng, D. Garlan, B. Schmerl, J. P. Sousa, B. Spitznagel, and P. Steenkiste, "Using Architectural Style as a Basis for System Self-Repair," *Proceedings of the Working IEEE/IFIP Conference on Software Architecture* (2002), pp. 45–59.
21. S. Shrivastava and S. Wheater, "Architectural Support for Dynamic Reconfiguration of Large Scale Distributed Applications," *Proceedings of the 4th International Conference on Configurable Distributed Systems* (May 1998), pp. 10–17.
22. P. Feiler and J. Li, "Consistency in Dynamic Reconfiguration," *Proceedings of the 4th International Conference on Configurable Distributed Systems* (May 1998), pp. 189–196.

*Accepted for publication September 16, 2002.*

**Keith Whisnant** *Center for Reliable and High-Performance Computing, University of Illinois at Urbana-Champaign, 1308 W. Main Street, Urbana, Illinois 61801 (electronic mail: kwhisnan@crhc.uiuc.edu).* Mr. Whisnant is currently a Ph.D. student at the University of Illinois. He holds a B.S. and an M.S. degree in computer engineering, also from the University of Illinois. While in graduate school, he has been the lead architect and system designer for the Chameleon ARMORs project, which uses a reconfigurable software infrastructure to provide a wide variety of fault-tolerant services to user applications. He has worked with the Jet Propulsion Laboratory in applying the ARMOR concepts to a software-implemented fault-tolerant environment for protecting parallel scientific applications that execute in space.

**Zbigniew T. Kalbarczyk** *Center for Reliable and High-Performance Computing, University of Illinois at Urbana-Champaign, 1308 W. Main Street, Urbana, Illinois 61801 (electronic mail: kalbar@*

*crhc.uiuc.edu).* Dr. Kalbarczyk is currently Principal Research Scientist at the Center for Reliable and High-Performance Computing in the Coordinated Science Laboratory of the University of Illinois at Urbana-Champaign. He holds a Ph.D. in computer science. After receiving his doctorate, he worked as Assistant Professor in the Laboratory for Dependable Computing at Chalmers University of Technology in Gothenburg, Sweden. Currently he is one of the leading researchers on the project to explore and develop approaches and techniques for providing an adaptive software infrastructure that allows different levels of availability requirements to be supported in a network environment. Dr. Kalbarczyk's research also involves developing automated fault/error injection techniques for validation and benchmarking of dependable and secure computing systems. He has served as a program co-chair of the International Performance and Dependability Symposium (IPDS), a track of the Conference on Dependable Systems and Networks (DSN 2002), and is regularly invited to work on the program committees of major conferences on design of fault-tolerant systems.

**Ravishankar K. Iyer** *Center for Reliable and High-Performance Computing, University of Illinois at Urbana-Champaign, 1308 W. Main Street, Urbana, Illinois 61801 (electronic mail: iyer@ crhc.uiuc.edu).* Professor Iyer is Director of the Coordinated Science Laboratory (CSL) at the University of Illinois at Urbana-Champaign, where he is George and Ann Fisher Distinguished Professor of Engineering. He holds appointments in the Department of Electrical and Computer Engineering and the Department of Computer Science. He is Co-Director of the Center for Reliable and High-Performance Computing at CSL. His research interests are in the area of reliable networked systems. He currently leads the Chameleon ARMORs project at the University of Illinois, which is developing adaptive architectures for supporting a wide range of dependability and security requirements in heterogeneous networked environments. Professor Iyer is a Fellow of the IEEE and the ACM, and an Associate Fellow of the American Institute for Aeronautics and Astronautics (AIAA). He has received several awards, including the Humboldt Foundation Senior Distinguished Scientist Award for excellence in research and teaching, the AIAA Information Systems Award and Medal for "fundamental and pioneering contributions towards the design, evaluation, and validation of dependable aerospace computing systems," and the IEEE Emanuel R. Piore Award "for fundamental contributions to measurement, evaluation, and design of reliable computing systems."

# autonomic computing:

## IBM's Perspective on the State of Information Technology

IBM®

the information technology industry loves to prove the impossible possible.

We obliterate barriers and set records with astonishing regularity. But now we face a problem springing from the very core of our success — and too few of us are focused on solving it.

More than any other I/T problem, this one — if it remains unsolved — will actually prevent us from moving to the next era of computing. Interestingly enough, it has little to do with the usual barriers that preoccupy us.

It's not about keeping pace with Moore's Law, but rather dealing with the consequences of its decades-long reign. It's not directly related to how many bits we can squeeze into a square inch, or how thinly we can etch lines in silicon. In fact, a continued obsession with the smaller/faster/cheaper triumvirate is really a distraction.

It's not a barrier of "machine intelligence," either, that threatens our progress. It has less to do with building "thinking machines" that embody the popular conception of artificial intelligence (AI) than automating the day-to-day functioning of computing systems. It may sound odd coming from the creators of Deep Blue, but we don't really need a better chess-playing supercomputer — or sentient machines and androids programmed to love and laugh — to overcome the largest obstacle standing in our way.

The obstacle is complexity. Dealing with it is the single most important challenge facing the I/T industry.

*It is our next Grand Challenge.*

We in the I/T industry continue to create increasingly powerful computing systems. Why? To make individuals and businesses more productive by automating key tasks and processes. *And for good reason:*

**in the evolution of humans and human society, automation has always been the foundation for progress.** Relegate life's mundane requirements to "automatically handled," and we free our minds and resources to concentrate on previously unattainable tasks. Few of us worry about harvesting the grain to grind the flour to bake bread — we buy it at a nearby store — or about how we'll connect with a friend halfway across the globe — we simply pick up the phone. **SEE FIGURES 1 & 2**

But evolution via automation also produces complexity as an unavoidable byproduct. *Computing systems especially have proved this true.*

Follow the evolution of computers from single machines to modular systems to personal computers networked with larger machines and an unmistakable pattern emerges: incredible progress in almost every aspect of computing — microprocessor power up by a factor of 10,000, storage capacity by a factor of 45,000, communication speeds by a factor of 1,000,000 — but at a price. Along with that growth has come increasingly sophisticated architectures governed by software whose complexity now routinely demands tens of millions of lines of code. Some operating environments weigh in at over 30 million lines of code created by over 4,000 programmers!



**FIGURE 1**

## Progress in Agriculture

**Nearly two centuries of innovations in automating manual tasks in agriculture have enabled efficiencies in both the production and yield of crops. Within this time frame farming as a percentage of the labor force decreased from 90 percent to 2.6 percent and labor hours to produce 100 bushels of wheat dropped from 300 hours to just 3 hours.** (Source: U.S. Department of Agriculture)

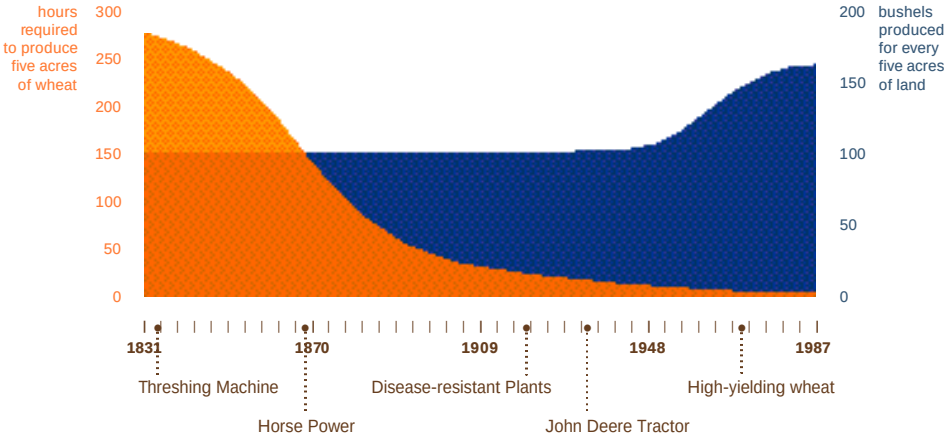
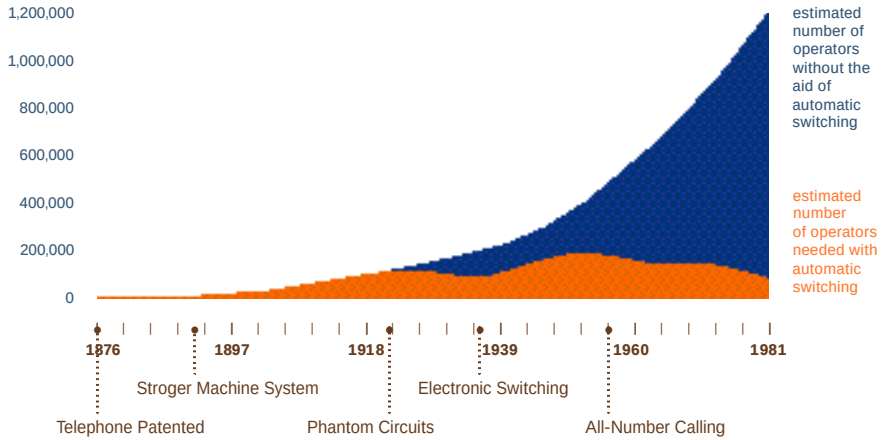



**FIGURE 2**

## Progress in Telephony (U.S. Only)

**AT&T/Bell System's implementation of an automated switching protocol in the 1920s allowed it to meet demand for telephones without outpacing the supply of human switchboard operators.** (Source: AT&T/Bell Systems)



The Internet adds yet another layer of complexity by allowing us to connect—some might say entangle—this world of computers and computing systems with telecommunications networks. In the process, the systems have become increasingly difficult to manage and, ultimately, to use—ask anyone who's tried to merge two I/T systems built on different platforms, or consumers who've tried to install or troubleshoot DSL service on their own.

In fact, the growing complexity of the I/T infrastructure threatens to undermine the very benefits information technology aims to provide. Up until now, we've relied mainly on human intervention and administration to manage this complexity. *Unfortunately, we are starting to gunk up the works.*



Consider this: at current rates of expansion, there will not be enough skilled I/T people to keep the world's computing systems running. Unfilled I/T jobs in the United States alone number in the hundreds of thousands. Even in uncertain economic times, demand for skilled I/T workers is expected to increase by over 100 percent in the next six years. Some estimates for the number of I/T workers required globally to support a billion people and millions of businesses connected via the Internet—a situation we could reach in the next decade—put it at over 200 million, or close to the population of the entire United States.

Even if we could somehow come up with enough skilled people, the complexity is growing beyond human ability to manage it. As computing evolves, the overlapping connections, dependencies, and interacting applications call for administrative decision-making and responses faster than any human can deliver. Pinpointing root causes of failures becomes more difficult, while finding ways of increasing system efficiency generates problems with more variables than any human can hope to solve.

Without new approaches, things will only get worse. Paradoxically, to solve the problem—make things simpler for administrators and users of I/T—*we need to create more complex systems.*

## *How will this possibly help?*

By embedding the complexity in the system infrastructure itself—both hardware and software — then automating its management. For this approach we find inspiration in the massively complex systems of the human body.

Think for a moment about one such system at work in our bodies, one so seamlessly embedded we barely notice it: *the autonomic nervous system*.

It tells your heart how fast to beat, checks your blood's sugar and oxygen levels, and controls your pupils so the right amount of light reaches your eyes as you read these words. It monitors your temperature and adjusts your blood flow and skin functions to keep it at 98.6°F. It controls the digestion of your food and your reaction to stress—it can even make your hair stand on end if you're sufficiently frightened. It carries out these functions across a wide range of external conditions, always maintaining a steady internal state called homeostasis while readying your body for the task at hand. SEE FIGURE 3

But most significantly, it does all this without any conscious recognition or effort on your part. This allows you to think about what you want to do, and not how you'll do it: you can make a mad dash for the train without having to calculate how much faster to breathe and pump your heart, or if you'll need that little dose of adrenaline to make it through the doors before they close.

It's as if the autonomic nervous system says to you, *Don't think about it — no need to. I've got it all covered.*

that's precisely how we need to build computing systems—an approach we propose as *autonomic computing.*

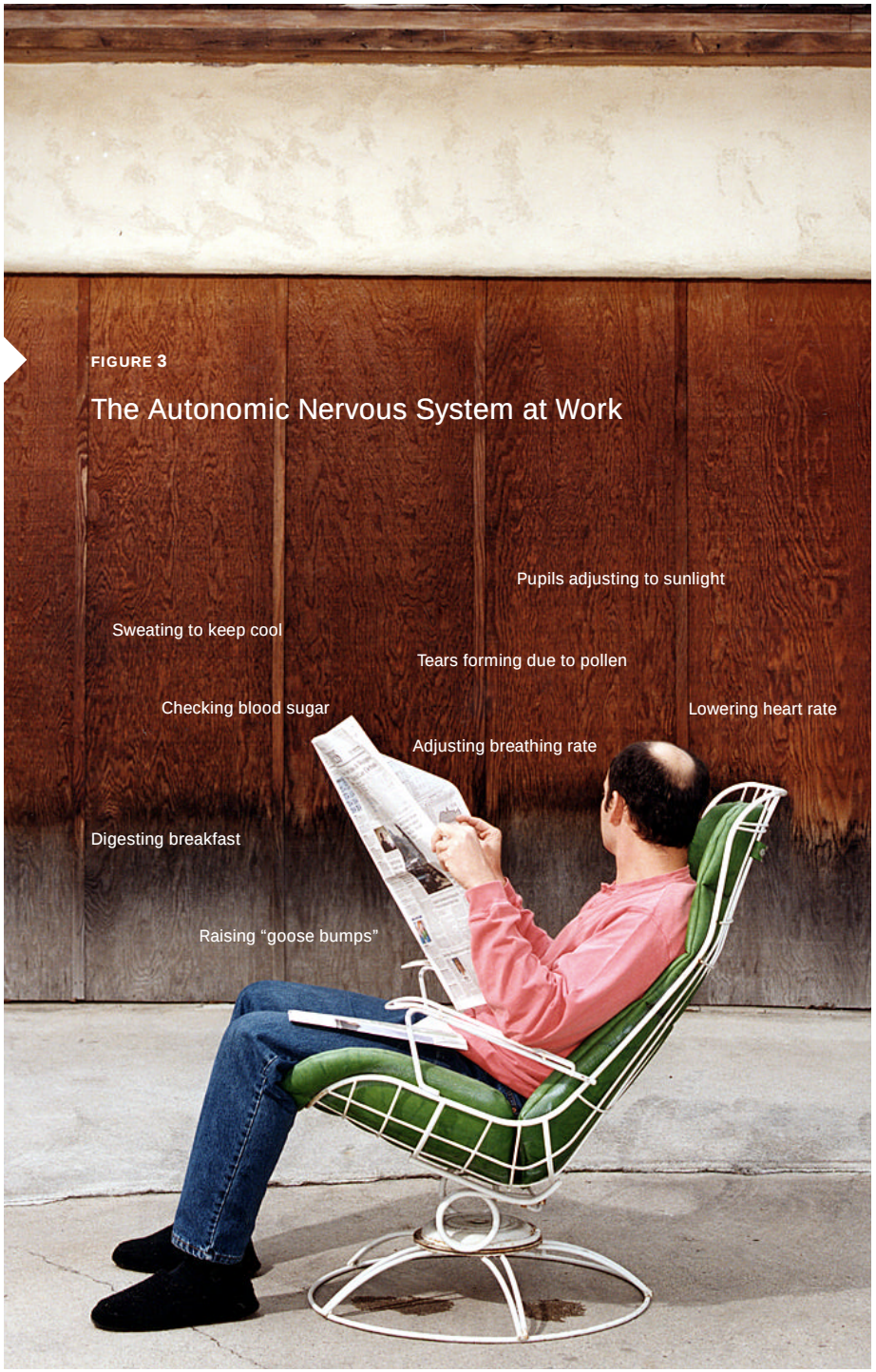



**FIGURE 3**

The Autonomic Nervous System at Work

it's time

to design and build computing systems capable of *running themselves*, adjusting to varying circumstances, and preparing their resources to handle most efficiently the workloads we put upon them. These autonomic systems must anticipate needs and allow users to concentrate on what they want to accomplish rather than figuring how to rig the computing systems to get them there.

*Why the urgency?* Consider some major challenges faced by organizations embracing new e-business technologies:



As a proliferating host of access devices becomes part of the corporate computing infrastructure, enterprises must transform both their I/T systems and the business processes to connect with employees, customers and suppliers. Not only must they manage desktops, workstations and PCs, but also PDAs, cell phones, pagers, etc. Annual compound growth of these devices is expected to exceed 38 percent over the next three years. Companies must also manage the very products they produce, such as network-enabled cars, washing machines, and entertainment systems, as part of this integrated "system," extending the system concept well beyond traditional corporate boundaries. This demands a reliable infrastructure that can accommodate rapid growth and hide system complexity from its users — the company's customers, employees and suppliers.

Emerging "Web Services" standards promise to make delivery of valuable services over the Internet possible. In one recent *InfoWorld* survey, close to 70 percent of respondents said they would develop Web Services strategies within the year, and roughly the same percentage felt Web Services likely to emerge as the next viable business model of the Internet. I/T services, in particular, are a likely candidate for delivery in a utility-like fashion, a trend we call e-sourcing. But such services cannot become widespread unless I/T systems become more automated and allow true economies of scale for e-sourcing providers. Customers also must gain enough confidence in this model to turn over critical business data and processes, confidence unlikely to develop if system reliability remains dependent on an inadequate supply of I/T workers.

The underlying technologies to enable greater automation of complex systems management are ripe for innovation. The emergence of XML and a host of new standards give us a glimpse of the glue we'll need to bind such self-governing systems, and advances in workload management and software agents promise possible incremental paths to autonomic computing.

But focusing on automating
the piece parts of computing systems
*will not be enough.*



*Think again of the functioning of the body's autonomic nervous system.*

it is the self-governing operation of the entire system, and not just parts of it, that delivers the ultimate benefit.

An increase in heart rate without a corresponding adjustment to breathing and blood pressure would be disastrous, just as a functioning brain stem would prove useless without an "always available" connection to the organs and muscles it directs.

So too with an autonomic computing system. Bringing autonomic capabilities to storage systems would certainly be an improvement, but if the computing systems that mine the data in those storage repositories become next to impossible to manage, that partial automation will not yield much overall benefit.

That's why we need a systemic approach that allows for coordination and automatic management across entire networks of computing systems—systems built on various platforms and owned (or even shared) by various entities. *Autonomic computing is thus a "holistic" vision* that will enable the whole of computing to deliver much more automation than the sum of its individually self-managed parts.

More is at stake than general system reliability or "ease of use" for I/T customers. Only when I/T complexity becomes embedded in the infrastructure, effectively eliminating its visibility from users, can the next wave of I/T-driven economic productivity occur. *In other words*, if we make it simpler for our customers to use our technology, new and even unpredictable applications of I/T will emerge. And more people will use them than ever before.

Both the Internet and the PC revolution accelerated computing's inevitable migration to the masses. But the vast majority of the earth's population has yet to touch a computer or benefit directly from its potential. For them to do so, interacting with computing systems will have to become much more natural. *How natural?*

As early as the 1960s, Captain Kirk and his crew were getting information from their computing systems by asking simple questions, and listening to straightforward answers—all without a single I/T administrator to be seen on the ship. Subtly, this has become the expectation of the masses: let me interact with computers as if I'm dealing with a person, but let the system have information processing capabilities no human could ever possess. While this goal may still sound futuristic,

*why not apply that thinking to* everyday business?



Why, for instance, should business owners have to spend so much time and money figuring out how to install and manage I/T systems? For some enterprise resource-planning systems, installation and customization fees can run several times the initial licensing cost of the system. All that should matter to the business owner is: *what does my business need to accomplish?*

ask i/t customers what they want, and they'll tell you: *let me concentrate on setting the business policy for, say, security, then have the system figure out the implementation details.* In much the same way programming languages have moved closer to natural language, companies should be able to frame their "instructions" for the computing system in plainspeak: watch my competitors, and if they're beating us in a particular market, adjust prices and the corresponding supply chain to support the new pricing.

To really benefit I/T customers, autonomic computing will need to deliver measurable advantage and opportunity by improving interaction with I/T systems and the quality of the information they provide, and enabling e-sourcing to be adopted as the future of I/T services delivery.

Then I/T customers will at last be able to focus entirely on the information services they want and "forget" about the systems providing them.

THE ACCOMPANYING FIGURES ILLUSTRATE THIS POTENTIAL.

FIGURE 4

**A large retail chain** with hundreds of outlets, a network of warehouses, delivery fleets, employee services, customer service call centers, web interfaces and more — an autonomic computing system manages all these distinct (and quasi-independent) I/T systems as one and provides integrated time-sensitive functionality, as well as *"always available"* access through web interfaces.

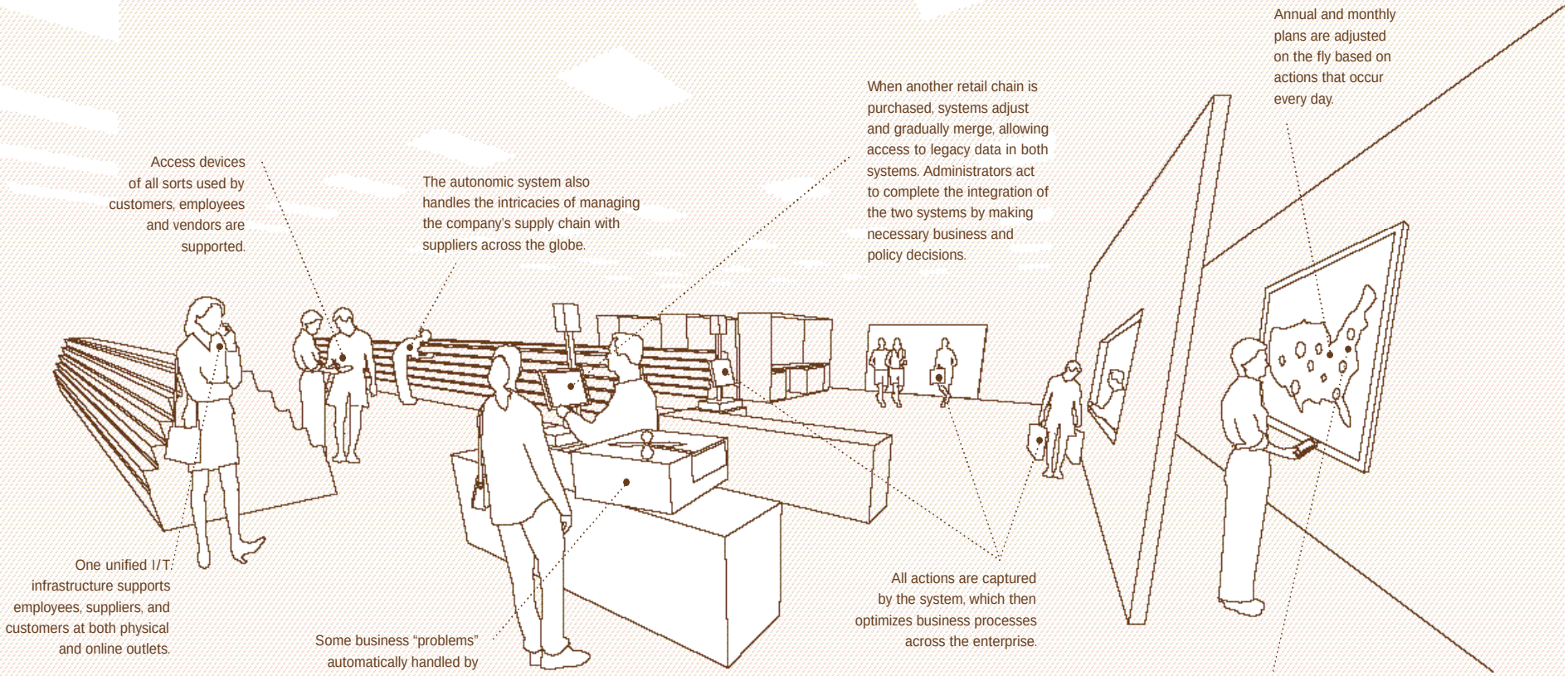
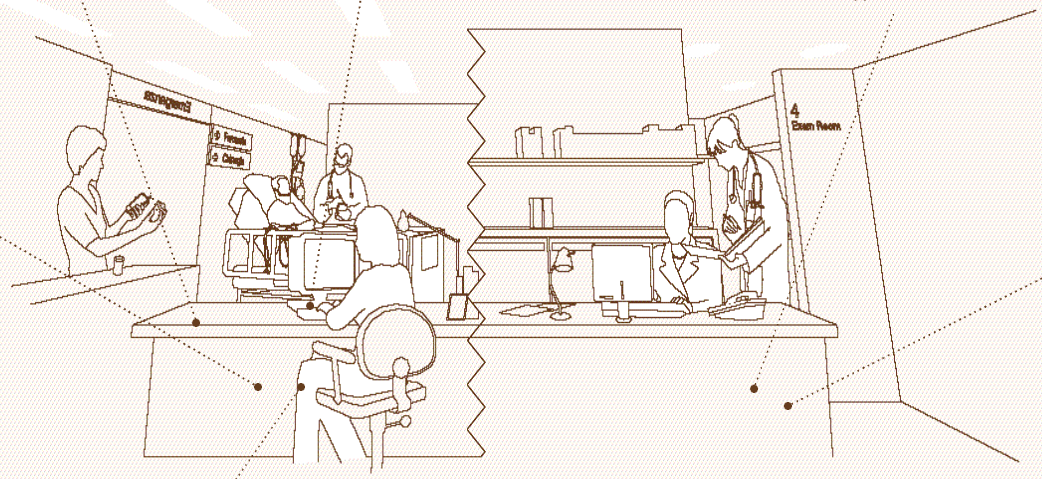


**FIGURE 5**

**A medical emergency in a foreign country** —dealing with such an urgent situation requires instant access and integration across multiple disparate systems. Autonomic systems management based on global medical standards enables distinct systems to act in a unified manner and exchange and integrate data. The result: a quick and accurate diagnosis as well as the best course for immediate life-saving treatment.




Even though the patient is initially unconscious, autonomic computing allows immediate access to overseas records, including complete family histories, x-rays and MRIs, which rule out some common diagnoses and indicate something more serious that requires surgery.

Doctors evaluate all current prescriptions and possible interactions as well as allergies, which alters plans for the type of anesthesia to be used.

Systems link to insurance records, obtain all necessary approvals and submit all forms.

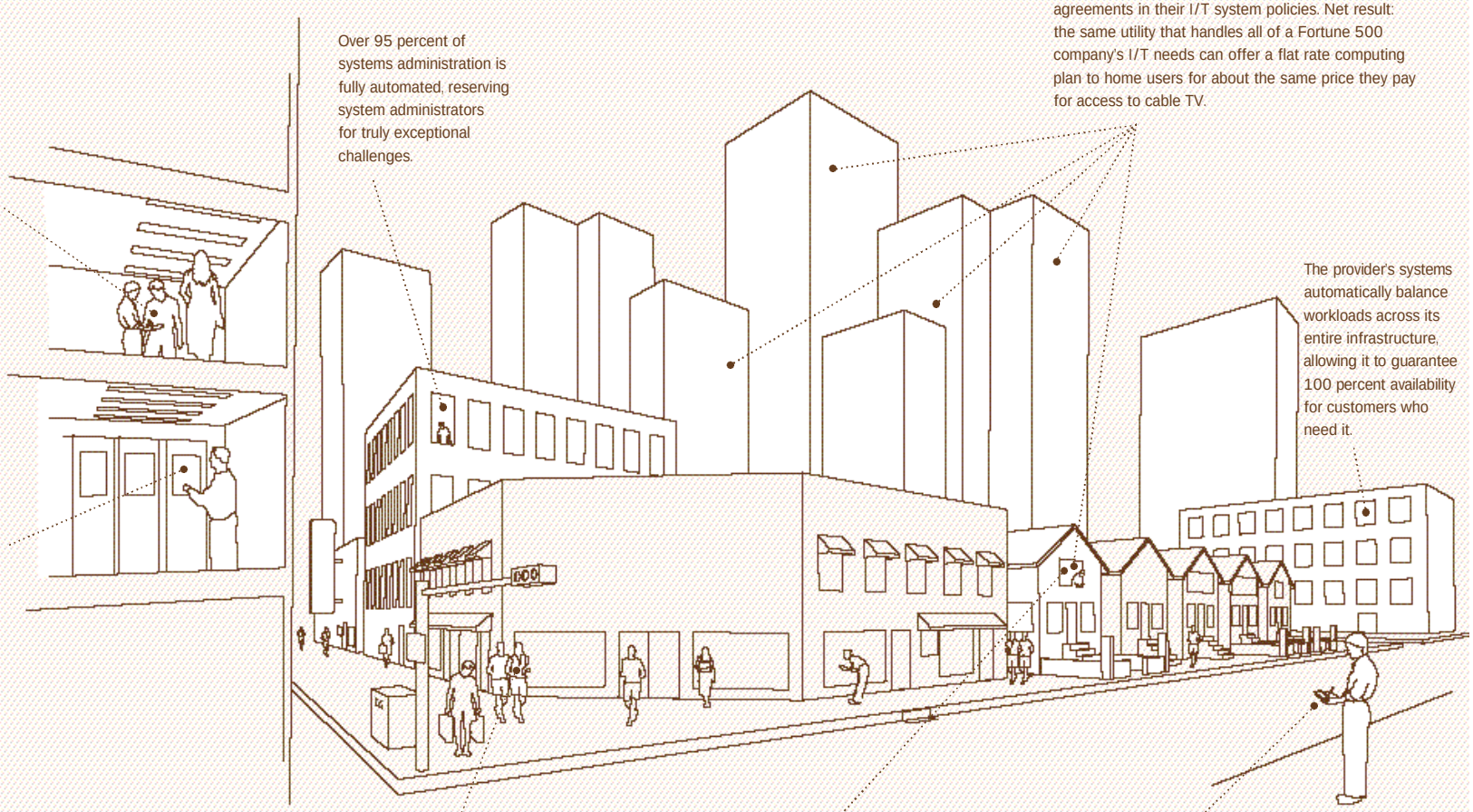
Patient's DNA record and known genomic data compared when evaluating potential treatments. Super-computing simulations run to ascertain best possible drug therapy, including a patient's unique systemic reaction to novel drugs.

Records accessed and appended only through trusted sources and with full protection of confidentiality based on standard medical processes embedded in software agents.

Reliability of the medical network — one which comprises many different systems across the globe belonging to different doctors and hospitals — uncompromised by local outages, storms or network traffic spikes.

FIGURE 6

**An e-sourcing provider** that services thousands of customers ranging from Fortune 500 companies to individuals—autonomic computing allows it to offer many ranges of service while profitably aggregating customers' varying demands across its global I/T infrastructure.



This level of functionality won't come easily, and may sound like the description of a panacea rather than an approach to the next era of computing. In fact, it's only the beginning of what we can imagine this next era will deliver. Unfortunately, that next era will never come if we continue managing systems and I/T complexity as we do today.

## so, if autonomic computing "systems" are the inevitable answer, what specifically are they?

First, let's define a "system." A system can be thought of as a collection of computing resources tied together to perform a specific set of functions. Following this logic, an individual server can constitute a system, as can a microprocessor containing varying integrated elements on a single chip — the so-called "system-on-a-chip" approach. These lower-level systems combine to form larger systems: multiprocessors to form servers, servers with storage devices and client or access devices to form networked systems, and so on. The principle of autonomic computing must govern all such systems—*i.e.*, at some level, they must be able to manage their own processes—although most of the defining elements of autonomic computing refer specifically to larger, higher-level systems.

Our bodies have a somewhat similar hierarchy of self-governance: from single cells to organs and organ systems (one such system being the autonomic nervous system). Each level maintains a measure of independence while contributing to a higher level of organization, culminating in the organism — us. We remain thankfully unaware, for the most part, of the daily management of it all, because these systems take care of themselves and only "escalate" to a higher level function when they need help.



So, too, with an autonomic computing system. In the end, its individual layers and components must contribute to a system that itself functions well without our regular interference to provide a simplified user experience. *Such a high-level system could be described as possessing at least* eight key elements *or characteristics.*

## 1

*To be* autonomic, a computing system needs to "know itself"—and comprise components that also possess a system identity.

Since a "system" can exist at many levels, an autonomic system will need detailed knowledge of its components, current status, ultimate capacity, and all connections with other systems to govern itself. It will need to know the extent of its "owned" resources, those it can borrow or lend, and those that can be shared or should be isolated.

Such system definition might seem simple, and when a "computer system" meant one room-filling machine, or even hundreds of smaller machines networked within the walls of one company, it was. But link those hundreds of computers to millions more over the Internet, make them interdependent, and allow a global audience to link back to those hundreds of computers via a proliferating selection of access devices — cell phones, TVs, intelligent appliances — and we have blurred the once-clear concept of a "system." Start allowing all those devices to share processing cycles, storage and other resources, add to that the possibility of utility-like leasing of computing services, and we arrive at a situation that would seem to defy any definition of a single "system."

But it's precisely this awareness at an overall system-wide level that autonomic computing requires. A system can't monitor what it doesn't know exists, or control specific points if its domain of control remains undefined.

To build this ability into computing systems, clearly defined policies embodied in adaptable software agents will have to govern a system's definition of itself and its interaction with I/T systems around it. These systems will also need the capacity to merge automatically with other systems to form new ones, even if only temporarily, and break apart if required into discrete systems.

## 2 *An* autonomic computing system must configure and reconfigure itself under varying and unpredictable conditions.

System configuration or "setup" must occur automatically, as must dynamic adjustments to that configuration to best handle changing environments.

Given possible permutations in complex systems, configuration can be difficult and time-consuming — some servers alone present hundreds of configuration alternatives. Human system administrators will never be able to perform dynamic reconfiguration as there are too many variables to monitor and adjust — multiply hundreds of alternatives by thousands of servers and other system devices — in too short a period of time — often minutes, if not seconds.

To enable this automatic configuration ability, a system may need to create multiple images of critical software, such as an operating system (a kind of software cloning), and reallocate its resources (such as memory, storage, communications bandwidth, and processing) as needed. If it is a globally distributed system, it will need to leverage its multiple images and backup copies to recover from failures in localized parts of its network. Adaptive algorithms running on such systems could learn the best configurations to achieve mandated performance levels.



## 3 *An* autonomic computing system never settles for the status quo — it always looks for ways to optimize its workings.

It will monitor its constituent parts and fine-tune workflow to achieve predetermined system goals, much as a conductor listens to an orchestra and adjusts its dynamic and expressive characteristics to achieve a particular musical interpretation.

This consistent effort to optimize itself is the only way a computing system will be able to meet the complex and often conflicting I/T demands of a business, its customers, suppliers and employees. And since the priorities that drive those demands change constantly, only constant self-optimization will satisfy them.

Self-optimization will also be a key to enabling the ubiquitous availability of e-sourcing, or a delivery of computing services in a utility-like manner. e-sourcing promises predictable costs and simplified access to computing for I/T customers. For providers of those computing services, though, delivering promised quality of service to its customers will require not only prioritizing work and system resources, but also considering supplemental external resources (such as subcontracted storage or extra processing cycles) similar to the way power utilities buy and sell excess power in today's electricity markets.

But to be able to optimize itself, a system will need advanced feedback control mechanisms to monitor its metrics and take appropriate action. Although feedback control is an old technique, we'll need new approaches to apply it to computing. We'll need to answer questions such as how often a system takes control actions, how much delay it can accept between an action and its effect, and how all this affects overall system stability.

Innovations in applying control theory to computing must occur in tandem with new approaches to overall systems architecture, yielding systems designed with control objectives in mind. Algorithms seeking to make control decisions must have access to internal metrics. And like the tuning knobs on a radio, control

points must affect the source of those internal metrics. Most important, all the components of an autonomic system, no matter how diverse, must be controllable in a unified manner.

## 4

## *An* autonomic computing system must perform something akin to healing — it must be able to recover from routine and extraordinary events that might cause some of its parts to malfunction.

It must be able to discover problems or potential problems, then find an alternate way of using resources or reconfiguring the system to keep functioning smoothly. Instead of "growing" replenishment parts, as our cells do, healing in a computing system means calling into action redundant or underutilized elements to act as replacement parts. (In a sense, this is akin to what our brain does when parts of it are damaged.)

Of course, certain types of "healing" have been a part of computing for some time. Error checking and correction, an over 50-year-old technology, enables transmission of data over the Internet to remain remarkably reliable, and redundant storage systems like RAID allow data to be recovered even when parts of the storage system fail.

But the growing complexity of today's I/T environment makes it more and more difficult to locate the actual cause of a breakdown, even in relatively simple environments. We see this even with personal computers — how many times is the "solution" to a problem "shut down, reboot and see if it helps"?

In more complex systems, identifying the causes of failures calls for root-cause analysis (an attempt to systematically examine what did what to whom and to home in on the origin of the problem). But since restoring service to the customer and minimizing interruptions is the primary concern, an action-oriented approach (determining what immediate actions need to be taken given current information available) will need to take precedence in an autonomic solution.

Initially, "healing" responses taken by an autonomic system will follow rules generated by human experts. But as we embed more intelligence in computing systems, they will begin to discover new rules on their own that help them use system redundancy or additional resources to recover and achieve the primary objective: meeting the goals specified by the user.



## 5

## *A* virtual world is no less dangerous than the physical one, so an autonomic computing system must be an expert in self-protection.

It must detect, identify and protect itself against various types of attacks to maintain overall system security and integrity.

Before the Internet, computers operated as islands. It was fairly easy then to protect computer systems from attacks that became known as "viruses." As the floppy disks used to share programs and files needed to be physically mailed or brought to other users, it took weeks or months for a virus to spread.

The connectivity of the networked world changed all that. Attacks can now come from anywhere. And viruses spread quickly — in seconds — and widely, since they're designed to be sent automatically to other users. The potential damage to a company's data, image and bottom line is enormous.

More than simply responding to component failure, or running periodic checks for symptoms, an autonomic system will need to remain on alert, anticipate threats, and take necessary action. Such responses need to address two types of attacks: viruses and system intrusions by hackers.

By mimicking the human immune system, a "digital immune system"—an approach that exists today—can detect suspicious code, automatically send it to a central analysis center, and distribute a cure to the computer system. The whole process takes place without the user being aware such protection is in process.

To deal with malicious attacks by hackers, intrusion systems must automatically detect and alert system administrators to the attacks. Currently, computer security experts must then examine the problem, analyze it and repair the system. As the scale of computer networks and systems keeps expanding and the likelihood of hacker attacks increases, we will need to automate the process even further. There won't be enough experts to handle each incident.

## 6 *An* autonomic computing system knows its environment and the context surrounding its activity, and acts accordingly.

This is almost self-optimization turned outward: an autonomic system will find and generate rules for how best to interact with neighboring systems. It will tap available resources, even negotiate the use by other systems of its underutilized elements, changing both itself and its environment in the process—in a word, adapting. This context-sensitivity includes improving service based on knowledge about the context of a transaction.

Such an ability will enable autonomic systems to maintain reliability under a wide range of anticipated circumstances and combinations of circumstances (one day perhaps covering even unpredictable events). But more significantly, it will enable them to provide useful information instead of confusing data. For instance, delivering all the data necessary to display a sophisticated web page would be obvious overkill if the user was connected to the network via a small-screen cell phone and wanted only the address of the nearest bank. Or a business system might report changes in the cost of goods immediately to a salesperson in the middle of writing a customer proposal, where normally weekly updates would have sufficed.

Autonomic systems will need to be able to describe themselves and their available resources to other systems, and they will also need to be able to automatically discover other devices in the environment. Current efforts to share supercomputer resources via a "grid" that connects them will undoubtedly contribute technologies needed for this environment-aware ability. Advances will also be needed to make systems aware of a user's actions, along with algorithms that allow a system to determine the best response in a given context.



## 7 *An* autonomic computing system cannot exist in a hermetic environment.

While independent in its ability to manage itself, an autonomic computing system must function in a heterogeneous world and implement open standards—in other words, an autonomic computing system cannot, by definition, be a proprietary solution.

In nature, all sorts of organisms must coexist and depend upon one another for survival (and such biodiversity actually helps stabilize the ecosystem). In today's rapidly evolving computing environment, an analogous coexistence and interdependence is unavoidable. Businesses connect to suppliers, customers and partners. People connect to their banks, travel agents and favorite stores—regardless of the hardware they have, or the applications they are using. As technology improves, we can only expect new inventions and new devices —and an attendant proliferation of options and interdependency.

Current collaborations in computer science to create additional open standards have allowed new types of sharing: innovations such as Linux, an open operating system; Apache, an open web server; UDDI, a standard way for businesses to describe themselves, discover other businesses and integrate with them; and from the Globus project, a set of protocols to allow computer resources to be shared in a distributed (or "grid-like") manner. These community efforts have accelerated the move toward open standards, which allow for the development of tools, libraries, device drivers, middleware, applications, etc., for these platforms.

Advances in autonomic computing systems will need a foundation of such open standards. Standard ways of system identification, communication and negotiation — perhaps even new classes of system-neutral intermediaries or "agents" specifically assigned the role of cyber-diplomats to regulate conflicting resource demands — need to be invented and agreed on.

8

*Perhaps* most critical for the user, an autonomic computing system will anticipate the optimized resources needed while keeping its complexity hidden.

This is the ultimate goal of autonomic computing: the marshaling of I/T resources to shrink the gap between the business or personal goals of our customers, and the I/T implementation necessary to achieve those goals — without involving the user in that implementation.

Today our customers must adapt to a computing system by learning how to use it, how to interact with it, and how to collect, compare and interpret the various types of information it returns before deciding what to do. Even custom-made solutions rarely



interact seamlessly with all a company's other I/T systems, let alone all its data and documents. While some aspects of computing have improved for general users — graphical interfaces, for instance, are far easier for most people to use than command prompts and their corresponding dictionaries of commands — tapping the full potential of entire I/T systems is still too difficult.

But does this mean autonomic computing systems must begin to possess human intelligence so as to anticipate, perhaps even dictate, a user's I/T needs? No. Think again of the analogy of our bodies, and in particular one aspect of the autonomic nervous system responsible for what's commonly known as the "fight or flight response."

When faced with a potentially dangerous or urgent situation, our autonomic nervous system anticipates the potential danger before we become aware of it. It then "optimizes" our bodies for a selection of appropriate responses — specifically, the autonomic nervous system triggers our adrenal glands to flood the body with adrenaline, a hormone that supercharges the ability of our muscles to contract, increases our heart rate and breathing, and generally constricts blood vessels to increase our blood pressure (while dilating those that feed key areas such as the skeletal muscles). The net result: our body is superbly prepped for action, but our conscious mind remains unaware of anything but the key pieces of information required to decide whether to stay and act (the "fight" response) or run for the hills.

An autonomic system will allow for that kind of anticipation and support. It will deliver essential information with a system optimized and ready to implement the decisions users make and not needlessly entangle them in coaxing results from the system.

**Realistically,** such systems will be very difficult to build and will require significant exploration of new technologies and innovations. That's why we view this as a Grand Challenge for the entire I/T industry. *We'll need to make progress along two tracks:*

## making individual system components autonomic, and achieving autonomic behavior at the level of global enterprise i/t systems.

That second track may prove to be extremely challenging. Unless each component in a system can share information with every other part and contribute to some overall system awareness and regulation, the goal of autonomic computing will not really be reached. So one huge technical challenge entails figuring how to create this "global" system awareness and management. *Or to put it another way,* how do we optimize the entire stack of computing layers as a whole? It's not something we currently know how to do.

**We know** there are also many interim challenges: how to create the proper "adaptive algorithms"—sets of rules that can take previous system experience and use that information to improve the rules. Or how to balance what these algorithms "remember" with what they ignore. We humans tend to be very good at the latter—we call it "forgetting"—and at times it can be a good thing: we can retain only significant information and not be distracted by extraneous data.

**Still another** problem to solve: how to design an architecture for autonomic systems that provides consistent interfaces and points of control while allowing for a heterogeneous environment. We could go on, as the list of problems is actually quite long, but it is not so daunting as to render autonomic computing another dream of science fiction.

*In fact, we're beginning to make progress in key areas.*

first,

many established fields of scientific study will contribute to autonomic computing. What we've learned in artificial intelligence, control theory, complex adaptive systems and catastrophe theory, as well as some of the early work done in cybernetics, will give us a variety of approaches to explore. Current research projects at laboratories and universities include self-evolving systems that can monitor themselves and adjust to some changes, "cellular" chips capable of recovering from failures to keep long-term applications running, heterogeneous workload management that can balance and adjust workloads of many applications over various servers, and traditional control theory applied to the realm of computer science, to name just a few.



Also,

some aspects of autonomic computing are not entirely new to the I/T industry. For instance, the protocols and standards used for forwarding packets of information across the Internet (the most well-known being TCP/IP) allow for some relatively simple functions, such as routing, to occur with little human direction. And since mainframe computers have historically been entrusted with important, "mission-critical" work for businesses and governments, they have had increasing levels of self-regulation and self-diagnosis built in. Such machines now boast "availability rates" — the percentage of time they are functioning properly — in the 99.999 percent range.

but this innovation needs to be taken to an entirely new level.

That's why at IBM we've reorganized our Research division around achieving this ambitious goal. And to accelerate the flow of this current innovation into today's hardware and software, we have undertaken Project eLiza: a major commitment of our server R&D budget earmarked for autonomic computing innovations.

this is
bigger than any *single*
i/t company.

It's a vision that requires the involvement of the top minds in the technology community. That's why we're forming an advisory board of leading academic and industry thinkers to help define our autonomic research agenda. We're also consulting with a large group of customers and I/T partners as part of our eLiza project to define a strategy for bringing autonomic innovations to products.

We call on our academic colleagues to drive exploratory work in autonomic computing. We propose that the research community recognize it as an important field of academic endeavor. We also call on our partners at government labs to collaborate with us on crucial projects in this area. We plan to fund a regular stream of academic grants to support research in this area, and we call on others in the I/T industry to do the same.



Finally, we call on the entire I/T industry to refocus its priorities on this essential goal. We must cooperate in developing the necessary standards and open interfaces to make this vision a reality. That's why we're working with Globus and the Grid Computing community to establish standards that will support an autonomic computing environment. *The days of pushing proprietary agendas and strategies are over. Our customers deserve better.*

We in the I/T industry have lingered too long in an era of over-specialization in which integration was just another specialty. We've made tremendous progress in almost every aspect of computing, but not enough in the one that now counts most: how to deal with the complexity generated by all that "smaller/faster/cheaper" focus.

In this heady rush we've risked losing sight of the people who buy I/T and who have come to depend on us for increased productivity and improvement in many aspects of their daily lives. We've made it unnecessarily difficult for them to tap the potential we've promised them. *It's time for a change.*

Autonomic computing represents both this change and the inevitable evolution of I/T automation. This next era of computing will enable progress and abilities we can barely envision today. But the best measure of our success will be when our customers think about the functioning of computing systems about as often as they think about the beating of their hearts.

Paul Horn, *Senior Vice President*
ibm research

## GLOSSARY

**Adaptive Algorithm**
An algorithm that can “learn” and change its behavior by comparing the results of its actions with the goals that it is designed to achieve.

**Algorithm**
A procedure, which can be written as a set of steps, for producing a specific output from a given input.

**Artificial Intelligence (AI)**
The capacity of a computer or system to perform tasks commonly associated with the higher intellectual processes characteristic of humans. AI can be seen as an attempt to model aspects of human thought on computers. Although certain aspects of AI will undoubtedly make contributions to autonomic computing, autonomic computing does not have as its primary objective the emulation of human thought.

**Autonomic**
1. Of, relating to, or controlled by the autonomic nervous system.
2. Acting or occurring involuntarily; automatic: an autonomic reflex.

**Autonomic Nervous System**
That part of the nervous system that governs involuntary body functions like respiration and heart rate.

**Catastrophe Theory**
A special branch of dynamical systems theory that studies and classifies phenomena characterized by sudden shifts in behavior arising from small changes in circumstances.

**Control Theory**
The mathematical analysis of the systems and mechanisms for achieving a desired state under changing internal and external conditions.

**Cybernetics**
A term derived from the Greek word for “steersman” that was introduced in 1947 to describe the science of control and communication in animals and machines.

**Feedback Control**
A process by which output or behavior of a machine or system is used to change its operation in order to constantly reduce the difference between the output and a target value. A simple example is a thermostat that cycles a furnace or air conditioner on and off to maintain a fixed temperature.

**Globus**
A collaborative academic project centered at Argonne National Laboratory focused on enabling the application of grid concepts to computing.

**Grand Challenge**
A problem that by virtue of its degree of difficulty and the importance of its solution, both from a technical and societal point of view, becomes a focus of interest to a specific scientific community.

**Grid computing**
A type of distributed computing in which a wide-ranging network connects multiple computers whose resources can then be shared by all end-users; includes what is often called “peer-to-peer” computing.

**Homeostasis**
A physiological constancy or equilibrium maintained by self-regulating mechanisms.

**Policy-based Management**
A method of managing system behavior or resources by setting "policies" (often in the form of "if-then" rules) that the system interprets.

**Project eLiza**
An initiative launched by IBM in April 2001 to integrate autonomic capabilities into its products and services, including servers, storage, middleware, and various kinds of services offerings. It is a core element of IBM's autonomic computing effort.

**Quality of Service (QoS)**
A term used in a Service Level Agreement (SLA) denoting a guaranteed level of performance (e.g., response times less than 1 second).

**Redundant Arrays of Independent Disks (RAID)**
A way of storing the same data in different places on multiple hard disks. Storing data on multiple disks can improve performance by balancing input and output operations. Since using multiple disks increases the mean time between failure, storing data redundantly also increases fault-tolerance.

**Service Level Agreement (SLA)**
A contract in which a service provider agrees to deliver a minimum level of service.

**Web Services**
A way of providing computational capabilities using standard Internet protocols and architectural elements. For example, a database web service would use web browser interactions to retrieve and update data located remotely. Web services use UDDI to make their presence known.


New Orchard Road, Armonk, NY 10504

This book is printed on recycled paper and is recyclable.



Printed in the U.S.A.
October 2001

Design: VSA Partners, Inc.

For more information about autonomic computing, please visit our website at
http://www.ibm.com/research/autonomic

# Expected Products or Implementation in near future

## Autonomic Computing for IBM eServer™ xSeries™

http://www-1.ibm.com/servers/autonomic/xseries.html

IBM eServer xSeries includes a rich portfolio of simply smart management tools designed to support IBM's autonomic computing initiative today. These tools have predictive and proactive features which automate IT tasks, requiring little or no human interaction. These intelligent capabilities are designed to deliver significant IT savings by avoiding unplanned system downtime and by reducing the time and labor associated with managing complex networked environments.

### Self-configuring features

- Dynamic partitioning
- Mass configuration support
- Automatic software/firmware update (BIOS, device drivers, etc.)
- Self-install and configuration of multiple, replicated, distributed systems (Remote Deployment Manager)
- Active™ PCI, memory
- IBM Director Configuration Wizards

### Self-healing features

- Memory mirroring
- Real Time Diagnostics
- Chipkill™ Memory, Hot Spare Memory
- Auto server restart
- Watchdog timer for automated reboot
- Flash Copy™
- RAID
- IBM Director:
    - Automated system health management
    - Software Rejuvenation to predict and avoid software failures
    - Capacity Manager resource bottleneck identification
    - Automated corrective action
- Electronic Service Agent - "call home" support

### Self-optimizing features

- Unattended, remote, replicated image deployment (Remote Deployment Manager)
- Automated group bottleneck detection & resolution
- Dynamic server clustering
- IBM Director (Automated Event Action Plans in response to alerts, automated task scheduler, process control tool)
- XpandOnDemand™ Scalability
- Active partitioning
- Active PCI Manager for optimizing I/O throughput

### Self-protecting features

- Predictive Failure Analysis with automated corrective action
- Automated FRU number reporting (failing part)
- SSL Web-based agent [manages Internet transmission security]
- LDAP [aids in the location of network resources]
- Kerberos [authenticates requests for service in a network]
- VPN
- Digital certificates [provides identity authentication]
- Encryption [prevents unauthorized use of data]

## Autonomic Computing for IBM eServer™ iSeries™

http://www-1.ibm.com/servers/autonomic/iseries.html

Many of the foundation technologies encompassed by autonomic computing manifest themselves in IBM eServer iSeries. From the day this advanced architecture was introduced to the marketplace it has contained many forward-looking autonomic technologies, such as the ability to automatically phone for service in anticipation of a failure, and the ability to automatically allocate data to all available disk drives while self-optimizing disk drive utilization. iSeries remains a leader by yielding a low overall total cost of ownership among platform choices. As iSeries has grown in performance and versatility, the autonomic advances have kept coming, including dynamic LPAR (Logical Partitioning) for efficient resource allocation, dynamic optimization of workloads and DB2, graphical tools for wizard-assisted setup and systems management, and Enterprise Identity Mapping for improved enterprise-wide security. iSeries -- it's "enterprise computing made simple."

### Self-configuring features

- CPU capacity upgrade on demand, providing instant access to reserve processors

- Hot plug disk, I/O
- SAN-like dynamic storage add for Windows (Intel) workloads (IXS, IXA)
- EZSetup Wizards, easing installation capabilities in servers and software
- iSeries Access for Wireless, providing systems management from anywhere
- Connect for iSeries, providing automated B2B connection services
- Enterprise Identity Mapping, improving enterprise-wide security and integrity
- Virtual I/O for Linux partitions and Windows (Intel) workloads
- Switchable auxiliary storage pools (independent ASP)
- NetServer (file/print server) support for Windows clients

## Self-healing features

- Auto-fix defective PTFs
- Chipkill Memory, dynamic bit steering
- Service director
- Performance Management/400 (PM/400)
- Tagged storage
- RAID
- Concurrent maintenance
- Dynamic IP takeover, clustering
- Independent auxiliary storage pools (switched disks)
- Electronic Service Agent - "call home" support
- Automatic performance adjuster
- Domino auto restart, clustering
- Agent-building learning environment (ABLE engine) for problem management

## Self-optimizing features

- Dynamic LPAR for OS/400 and Linux
- Adaptive e-transaction services
- Integrated database optimizer
- Automatic Index Advisor for DB
- Graphical database monitor
- Dynamic load balancing for disk capacity and arm utilization
- Single-level storage, Expert Cache option
- Quality of service
- Global resource manager, heterogeneous workload manager
- Automatic performance management/workload balancing

### Self-protecting features

- Self-protecting object-based kernel
- Digital object tagging, automatic object retranslation (virus removal)
- Tagged storage, RAID, ECC, dual line cords
- IP takeover
- Enterprise identity mapping
- Built in support for Kerberos [authenticates requests for service in a network], LDAP [aids in the location of network resources], SSL [manages Internet transaction security], VPN [manages Internet transmission security], Digital Certificates [provides identity authentication]

## Autonomic Computing for IBM eServer™ pSeries™

http://www-1.ibm.com/servers/autonomic/pseries.html

Today's IBM eServer pSeries offers some of the most advanced self-management features for Unix servers on the market today. In our most recent announce included items like: LPAR and dynamic LPAR, workload manager, Reliable Scalable Cluster Technology (RSCT), and a Self Protecting Kernel. Kerberos V5 Authentication, Automatic System & Etherchannel Failover, make the IBM eServer pSeries is one of the most reliable Unix system on the marketplace as well as one of the simplest to use and manage.

### Self-configuring features

- Virtual IP address (VIPA)
- IP multipath routing
- Microcode discovery services/inventory scout
- Hot-swappable disks
- Hot-swap PCI
- Wireless/pervasive system configuration
- TCP explicit congestion notification

### Self-healing features

- Multiple default gateways
- Automatic system hang recovery
- Automatic dump analysis & e-mail forwarding
- Ether channel automatic failover
- Graceful processor failure detection & failover

- HACMP & HAGeo
- First failure data capture
- Chipkill ECC Memory, dynamic bit steering
- Memory scrubbing
- Automatic, dynamic de-allocation (processors, L2/L3 cache, LPAR, PCI buses)
- Electronic Service Agent - "call home" support

### Self-optimizing features

- Static LPAR
- Dynamic LPAR
- Workload manager enhancement
- Extended memory allocator
- Reliable, scalable cluster technology (RSCT)
- PSSP cluster management

### Self-protecting features

- Kerberos V5 Authentication [authenticates requests for service in a network]
- Self-protecting kernel
- SecureWay LDAP directory integration [LDAP aids in the location of network resources]
- SSL [manages Internet transmission security]
- DigitalCertificates
- Encryption [prevents unauthorized use of data]

**Ex 1) ☞ See the Appendix I**

## Autonomic Computing for IBM eServer™ zSeries™

http://www-1.ibm.com/servers/autonomic/zseries.html

Today's IBM eServer zSeries offers some of the most advanced autonomic computing features available on the market. This includes features like CPU Sparing, dynamic LPAR as well as Intelligent Resource Director for dynamic workload and resource management facilities for the most demanding e-business environments. Some of the recent improvements include: setup wizards for VPN, IRD and security planner, security and clustering for improved "self configuration capabilities", Enterprise Identity mapping for improved enterprise-wide security and, intelligent resource director for improved resource utilization and management across a range of e-business workloads.

## Self-configuring features

- msys for Setup (for Parallel Sysplex, TCP/IP, USS, LE, ISPF,...)
- z/OS wizardry [enables self-installation capabilities] including new support for Intelligent Resource Director and security planner
- Capacity upgrade CPU [provides instant access to additional processors or servers], memory, I/O
- Customer Initiated Upgrade
- Automatic hardware detection/configuration
- Automatic communication configuration

## Self-healing features

- msys for Operations
- Concurrent memory upgrade (CoD)
- Capacity backup
- CPU, memory (ECC), I/O, network
- Sysplex CF duplexing
- Concurrent maintenance
- Dynamic Virtual IP takeover and takeback
- Dynamic disk balancing
- CF structure duplexing
- System automation for OS/390
- Electronic Service Agent - "call home" support
- Geographically Dispersed Parallel Sysplex
- Fault tolerant cache hierarchy

## Self-optimizing features

- IRD (Intelligent Resource Director) extensions (non z/OS partitions-CPU (Linux), I/O, server-wide) [allows dynamic resource allocation on zSeries servers]
- Dynamic LPAR, WLM LPAR extensions for Linux
- Parallel Sysplex Extensions- Sysplex Distributor, CP Coupling
- BIND9 DNS-DNS BIND Version 9.0 on z/OS
- z/OS workload manager (CPU, memory, I/O; TCP/IP QOS, Web request management, and batch initiator balancing)

## Self-protecting features

- LPAR, Certifications: FIPS, I40-1 Level 4, EAL4, ZKA, ICSA Labs 1.0A

- Intrusion detection-IDS, PKI
- Hardware cryptographic adapters
- Digital certificates [provides identity authentication]
- SSL and TLS [manages Internet transmission security], Kerberos [authenticates requests for service in a network], VPN, encryption
- Tivoli Policy Director
- LDAP [aids in the location of network resources]
- HiperSockets™

## Tivoli Software

http://www-3.ibm.com/autonomic/tivoli.shtml

IBM Tivoli software technology management software from IT systems. Used by 96% of Fortune 500 companies, Tivoli products make it easy to manage hundreds of thousands of separate devices without sacrificing productivity, security, or performance.

### Autonomic Computing Capabilities

Tivoli delivers a variety of autonomic computing capabilities based on the four characteristics of self-managing systems:

- **Self-configuring**
  - IBM Tivoli Storage Manager
  - IBM Tivoli Public Provisioning Infrastructure (with VeriSign)
- **Self-healing**
  - IBM Tivoli Switch Analyzer
  - IBM Tivoli Business Systems Manager
  - IBM Tivoli System Automation for OS/390®
  - IBM Tivoli Storage Resource Manager
- **Self-optimizing**
  - IBM Tivoli Workload Scheduler
  - IBM Tivoli Storage Manager
  - IBM Tivoli Analyzer for Lotus® Domino™
- **Self-protecting**
  - IBM Tivoli Access Manager
  - IBM Tivoli Identity Manager
  - IBM Tivoli Access Management Services (with VeriSign)

## Features and Benefits

- **Configuration management**

  Enables integrated inventory and software distribution

- **Storage resource management**

  Scans and discovers storage resources in the SLA management

  Manages Service Level Agreements (SLA breaches

- **Identity management**

  Enables centralized identity management and integrated, automated workflow with business processes

- **Monitoring capabilities**

  Provides tailored automation using templates applied against a resource model engine

- **Storage management**

  Enables automatic domain configuration and file system identification, storage-pool restoration and correction, backup and restore optimization, and automated protection of enterprise data across heterogeneous storage environments

- **Risk management**

  Offers self-protecting core technology to shield

## Next steps in autonomic computing

Tivoli software from IBM is delivering autonomic computing capabilities to customers today, and will continue developing advanced self-managing technologies to enable a more rapid return on IT investments than ever before.

### Ex 1) Tivoli Distributed Monitoring

Tivoli Distributed Monitoring provides an efficient, reliable, automated way to ensure user access to distributed applications and computing resources. Designed specifically for distributed computing, Tivoli Distributed Monitoring offers a consistent, easy and centralized way to group and monitor key computing resources.

This self-optimizing monitoring checks system resources and services while proactively detecting and automatically correcting potential problem situations for your business.

### Ex 2) Tivoli Enterprise Console

The Tivoli Enterprise Console (TEC) provides a centralized point of control that keeps your I/T staff in close and efficient control of events happening across all systems in your distributed environment. The TEC processes and responds to the thousands of events and alarms that occur daily from network devices, hardware systems, relational database management systems and Tivoli partner and customer applications.

The TEC processes and correlates network and system events and automatically initiates corrective actions,

enabling businesses to not only identify failures, but proactively avoid them, because indications of system resource problems can be seen hours before a failure occurs.

### Ex 3) Tivoli Storage Manager

This self-protecting storage manager helps customers automate network backup and efficiently manage storage while providing disaster recovery functions.

Tivoli Storage Manager will protect and manage your business information in an enterprise-wide Storage Area Network and traditional network environment. It includes a host of optional products offering an end-to-end scalable solution spanning laptops to mainframes on over 35 platforms.

### Ex 4) Tivoli Software Distribution

Tivoli Software Distribution focuses on deploying software on an enterprise scale. It is a key solution for customers who need to rapidly and efficiently deploy complex applications to multiple locations from a central point.

It enables businesses to dispatch software and updates globally in real time -- even assuring software delivery to systems that become disconnected.

### Ex 5) Tivoli Monitoring Resource Model Builder

http://www.alphaworks.ibm.com/tech/rmbuilder

The Tivoli Monitoring Resource Model Builder is a programming tool for creating, modifying, debugging, and packaging resource models for use with IBM Tivoli Monitoring products. New for Version 1.1.0 is a user interface based on WebSphere Studio Workbench, along with increased documentation and coding examples. The new Resource Model Builder makes it easier to specify events, parameters, and thresholds for existing and new resource models in order to identify server and system problems in real time, notify administrators, and take autonomic corrective action.

## Lotus Software

http://www-3.ibm.com/autonomic/lotus.shtml

Lotus software from IBM focuses on the human side of business, delivering solutions that empower people to collaborate, learn and make the most of their collective knowledge wherever they work -- in their offices, at home or on the road. With flexible, security-rich solutions that are open and reliable, Lotus has reinvented the way thousands of organizations around the world do business.

Our flagship products -- IBM Lotus Domino® -- are the rich products that introduced "groupware" to the business lexicon over a decade ago. With over 95 million users worldwide, we are a recognized leader and

innovator in the exciting world of messaging and collaboration. We view autonomic computing as a critical competitive point-of-differentiation that will help systems focus on the technology side of business, allowing people to focus on the human side of business.

### Autonomic Computing Capabilities

Release 6 of Lotus Notes and Domino(ND6) delivers a variety of autonomic computing capabilities based around the four characteristics of self-managing systems:

- **Self-configuring**
  Domino Administrator offers automatic propagation of design elements, new activity tracking capabilities, event handlers and automatic updating of directory elements
- **Self-healing**
  Tivoli Analyzer for Domino offers predictive capabilities
- **Self-optimizing**
  Tivoli Analyzer for Domino provides clustered server load balancing and distributed application processing to increase the productivity of administrative staff and enhance end user satisfaction
- **Self-protecting**
  ND6 delivers secure authentication and access control management, comprehensive encryption for all communications, and extensive support for security standards, protocols, and digital signatures

### Next steps in autonomic computing

The autonomic computing features in IBM Lotus Notes/Domino 6 are just the beginning. The vision to provide more extensive autonomic computing features that allow people to focus on collaborative high-value activities while the systems manage more and more of their own functions will unleash the true power of collaborative computing.

**Ex 1) Lotus Discovery Server**

http://www.lotus.com/products/discserver.nsf

The Discovery Server extracts, analyzes and categorizes structured and unstructured information to reveal the relationships between the content, people, topics and user activity in an organization. Through this expertise profiling and content discovery, the server uncovers organization know-how in terms of where things are, who knows what, what is relevant, and which subjects generate the most interest and interactivity.

This is an advanced data search and expertise location solution designed to help customers maintain and grow their knowledge capital.

## WebSphere Software

http://www-3.ibm.com/autonomic/websphere.shtml

The IBM WebSphere software platform expands your business opportunities and productivity with a world-class infrastructure ready for the next chapter in e-business. Version 5 of the WebSphere Application Server and WebSphere Studio products deliver:

- **A comprehensive build to integrate platform**
  Improve time-to-value by building new integration-ready applications that leverage your existing software assets
- **A highly integrated application development environment**
  Maximize your return on investment and reduce labor costs with superior developer productivity
- **Agile deployment and administration**
  Reduce cost of ownership and minimize startup investment with highly productive and flexible administration, deployment and management services
- **Intelligent end-to-end application optimization**
  Create competitive advantage and optimize price/performance while meeting the changing demands of dynamic e-business with industry leading reliability, availability, scalability, performance and security

## Autonomic Computing Capabilities

The WebSphere software platform delivers a variety of autonomic computing capabilities based on the four characteristics of self-managing systems.

- **Self-configuring**
  Lowers the cost of ownership and minimizes startup investment with highly productive and flexible administration, deployment and management services
- **Self-healing**
  Offers built-in resiliency to handle unpredictable high volume loads with high availability, even if hardware failures occur
- **Self-optimizing**
  Creates competitive advantage and optimizes price/performance for customers while also meeting the changing demands of dynamic e-business with industry-leading performance
- **Self-protecting**
  Delivers policy-based security and workload management to ensure:
  - Only authorized users access application resources
  - Consistent end-to-end application integrity is maintained

## Autonomic Computing Features and Benefits

- **Adapt to multiple deployment options**
  Dramatically reduces configuration complexity

- **Plug-and-play applications**
  Updates applications without having to stop the system, allowing you to continue working during updates
- **Problem analysis and recovery**
  Identifies patterns of well-known problems and makes corrective recommendations for quick recovery from problems
- **Application profiling**
  Applies system resources at runtime to optimize application execution based on usage profiles
- **Workload management**
  Optimizes workload and provides failover by enabling tasks to be distributed to different machines according to their capacity
- **Denial of service protection**
  Provides circuit-breaker capability for a clustered system, preventing a DoS attack from flooding the system with requests

### Next steps in autonomic computing

With autonomic computing capabilities built into the core foundation of WebSphere Application Server, it is an integral part of IBM's autonomic computing architecture and will yield even greater value in the future.

**Ex 1) WebSphere Site Analyzer**

http://www-3.ibm.com/software/sysmgmt/products/web-site-analyzer.html

IBM site analyzer provides analysis for enterprise website visitor traffic, visitor behavior, site usage, site content and site structure. It helps you make factual e-business decisions regarding website activity.

The analyzer helps I/T managers get to the root cause of a database error by identifying patterns and proactively contacting the administrator for a fix, rather than creating a customer call. The next step will be that when things fail, the fix will be applied automatically.

## IBM DB2 Universal Database

http://www-3.ibm.com/autonomic/db2.shtml

IBM DB2 Universal Database software is the worldwide market share leader in the industry and marks the next stage in the evolution of the relational database. DB2 is the industry's first multimedia, Web-ready relational database management system delivering leading capabilities in manageability, reliability, performance, and scalability. DB2 is the database of choice for customers and partners developing and deploying critical solutions. There are more than 60 million DB2 users from 400,000 companies worldwide relying on IBM data management technology.

As the scope of relational database functions has expanded in recent years, the complexity of database systems has also grown. The added complexity and the increase in data size — now frequently into tens of terabytes — have increased the burden on database administrators. The combination of increased data volumes, larger systems, and increased function has motivated the need for autonomic capability within database management systems in order to reduce cost of ownership and to enable databases to operate in environments with limited access to skilled administration personnel.

### Autonomic Computing Capabilities

DB2 Universal Database has long established the bar for self-optimization in the database industry with its advanced query optimization capabilities. With Version 8.1, DB2 expands its autonomic computing foundation with key new capabilities that deliver value today through reducing complexity in deploying, managing, and configuring relational databases:

- **DB2 Health Monitor**
  Provides out-of-the box monitoring of key DB2 performance and reliability metrics and offers expert advice on resolving any problems that may occur
- **DB2 Configuration Advisor**
  Allows even a novice user to achieve a configuration that previously would have taken an expert days to accomplish

### Next steps in autonomic computing

The current set of features in DB2 Universal Database only scratches the surface of the larger goal of complete autonomic computing for relational databases. Future releases of DB2 Universal Database will feature key infrastructure capabilities that will allow DB2 Universal Database to evolve to the next level of system automation and self-control by allowing a large set of operations, configuration changes, and maintenance utilities to run concurrently with online systems, and provide enhanced monitoring and reporting of system activity and resource utilization.

These monitoring and online capabilities have set the stage for research in adaptive resource control, dynamic tuning, automatic maintenance, and adaptive query access plan refinement. Development continues as well in enhanced physical database design technology, self-protecting, and self-healing technology.

## IBM Total Sotrage

http://www-3.ibm.com/autonomic/storage_sys.shtml

IBM total storage can help you address your storage challenges regardless of your company's size with:

- A portfolio of products that offer scalability, reliability, security and performance together with cross-platform

compatibility and provide cross-platform compatibility backed by industry-leading warranties

- A network of service personnel committed to providing timely and helpful global technical support
- Solutions that can address all your infrastructure needs, including:
    - Storage consolidation
    - Data protection
    - Disaster tolerance
    - Data sharing and access

## Autonomic Computing Capabilities

Autonomic storage is a composite set of autonomic computing capabilities that grows from making individual devices intelligent, aggregating those devices by their role, and connecting the entire network to business priorities. What emerges is storage that reflects the four key characteristics of self-managing systems.

Rather than being a specific technology breakthrough, autonomic storage is achieved through the gradual accumulation of many advanced technologies. Today, IBM offers a variety of storage products that include numerous autonomic computing functions. For example:

- **IBM TotalStorage Enterprise Storage Server™ (ESS)**
  Offers predictive failure analysis and auto configuration/rebuild
- **IBM ToralStorage FAStT Storage Servers**
  Provides LUN masking and partitioning as well as dynamic RAID-level migration and segment size changes
- **IBM TotalStorage Network Attached Storage (NAS)**
  Supports cluster failover/failback and predictive failure analysis with automated corrective actions
- **IBM TotalStorage Virtual Tape Server(VTS)**
  Includes space management and peer-to-peer copy
- **IBM TotalStorage LTO Ultrium Scalable Tape Library**
  Offers automatic calibration of hardware, and auto detection and identification of all features during configuration
- **IBM TotalStorage LTO Ultrium UltraScalable Tape Library**
  Includes logical library partitioning
- **IBM Tivoli Storage Resource Manager**
  Delivers a set of automated tools that address multiple aspects of the storage infrastructure

## Next steps in autonomic computing

**Storage Software**

Storage software for autonomic storage includes providing options for storage administrators to automate tasks. We plan to offer products that provide:

- Policy-based storage management provisioning
- Transparent data movement between volumes and storage pools
- Transparent addition, deletion, and redeployment of storage
- Auto-failover and failback of virtualization nodes

IBM intends to deliver a comprehensive set of storage software solutions that address the issues of storage management for block devices, files, and management of storage networks.

In addition, we plan on introducing a common file system specifically designed for storage networks based on our Storage Tank™ technology initiative. The Storage Tank design helps migrate intelligence from individual servers and onto the storage network where it can be available to all application servers on that network, helping to automate routine and error prone tasks using policy-based automation.

**Open standards-based management**

IBM TotalStorage products will come with basic management tools and will be manageable by IBM management products through the use of SNIA Bluefin standards. We plan to converge SNA and NAS environments for customers who need both and simplify data sharing and management.

We believe that standards efforts centered on SNIA will become highly accepted by the industry. Our products are being designed to support management through the use of open standards. We intend to provide industry standard interfaces for our virtualization and Storage Tank offerings. This will permit these products to be managed by the IBM management software products that support open standards.

## DataCase, Takmi (Call Center, Service, Repair, TCO reduction)

DataCase is a keyword search, question and answer technology that acts as an expert system for call centers. TAKMI (Text Analysis and Knowledge Mining) is text mining technology that has a graphical user interface to display the results of analysis. It indicates information such as word frequency, frequency of associated words, topic trend, and so on. Like DataCase, TAKMI has also been used by IBM PC help centers.

☞ **See the Appendix Ⅱ**

## Discovery Link

http://www-3.ibm.com/solutions/lifesciences/discoverylink.html

A system for integrated access to life sciences data sources. IBM builds upon its key strengths in storage and retrieval technologies to assist scientists and accelerate the drug discovery process with this new middleware solution.

DiscoveryLink will help to dramatically increase R&D productivity with single-query access to existing databases, applications, and search engines.

## Intelligent Agent: ABLE

http://alphaworks.ibm.com/aw.nsf/frame?ReadForm&/aw.nsf/techmain/1BD82BF8F90CB6F1882568D3005BA561

The Agent Building and Learning Environment (ABLE) is a Java framework and toolkit for the construction and deployment of intelligent agents. These intelligent algorithms understand a pattern (what you are trying to accomplish), figure out what needs to be done, and control the system.

In the ABLE framework, an agent is an autonomous software component. It could be running on its own thread of control or could be called synchronously by another agent or process either through a direct method call or by sending an event.

☞ **See the Appendix III**

## OptimalGrid

http://www.alphaworks.ibm.com/tech/optimalgrid

OptimalGrid is a research prototype of a grid-enabled collaboration framework, sophisticated management infrastructure, and problem-solving environment for grid computing that is designed to hide complexities of partitioning, distributing, and load balancing.

The goal of OptimalGrid is to simplify the creation and management of connected parallel applications on the grid by including important autonomic grid functionality in a middleware layer. Ideally, developers should be able to create grid-enabled parallel applications without themselves becoming experts in grid or high-performance computing. Grid applications should be able to automatically reconfigure themselves in the response to dynamic changes in the grid environment.

☞ **See the Appendix IV**

## Log and Trace Preview for Autonomic Computing

http://www.alphaworks.ibm.com/tech/logandtrace

The Log and Trace Preview for Autonomic Computing is an Eclipse-based tool that enables viewing, analysis, and correlation of log files generated by IBM WebSphere® Application Server, IBM HTTP Server, IBM DB2 Universal Database, and Apache HTTP Server.

This tool makes it easier and faster for developers and support personnel to debug and resolve problems within multi-tier systems by converting heterogeneous data into a common event model and by providing a specialized visualization and analysis of the data. The Log and Trace Preview for Autonomic Computing works with ISV application plug-ins.

## Emerging Technologies Toolkit

http://www.alphaworks.ibm.com/tech/ettk

The ETTK is a software development kit for designing, developing, and executing emerging autonomic and grid-related technologies and Web services. The ETTK provides an environment in which to run emerging technology examples that showcase recently announced specifications and prototypes from IBM's emerging technology development and research teams. In addition, it provides introductory material to help developers easily get started with development of autonomic technologies, Web services, and grids. The ETTK evolved from the package known as the Web Services Toolkit (WSTK). With the renaming of the WSTK package to ETTK, the scope of technologies included within the package has been expanded. The new toolkit consolidates related technologies from various IBM development and research labs. The demos and functions of the ETTK run on both Linux and Windows® operating systems.

# Dynamic reconfiguration: Basic building blocks for autonomic computing on IBM pSeries servers

by J. Jann
L. M. Browning
R. S. Burugula

A logical partition in an IBM pSeries™ symmetric multiprocessor (SMP) system is a subset of the hardware of the SMP that can host an operating system (OS) instance. Dynamic reconfiguration (DR) on these logically partitioned servers enables the movement of hardware resources (such as processors, memory, and I/O slots) from one logical partition to another without requiring reboots. This capability also enables an autonomic agent to monitor usage of the partitions and automatically move hardware resources to a needy OS instance nondisruptively. Today, as SMPs and nonuniform memory access (NUMA) systems become larger and larger, the ability to run several instances of an operating system(s) on a given hardware system, so that each OS instance plus its subsystems scale or perform well, has the advantage of an optimal aggregate performance, which can translate into cost savings for customers. Though static partitioning provides a solution to this overall performance optimization problem, DR enables an improved solution by providing the capability to dynamically move hardware resources to a needy OS instance in a timely fashion to match workload demands. Hence, DR capabilities serve as key building blocks for workload managers to provide self-optimizing and self-configuring features. Besides dynamic resource balancing, DR also enables Dynamic Capacity Upgrade on Demand, and self-healing features such as Dynamic CPU Sparing, a winning solution for users in this age of rapid growth in Web servers on the Internet.

One of the cardinal features of an autonomic component in an information technology (IT) infrastructure is the ability of the component to adapt itself smoothly to changes in its environment. Endowing a computing system with this self-management feature often translates to the implementation of self-protecting, self-healing, self-optimizing, and self-configuring algorithms and subcomponents. Because the primary role of an operating system (OS) is to manage the physical resources of a computer system so as to optimize the performance of its applications (including middleware, which consists of applications from the perspective of the OS), an OS supporting autonomic computing[1] needs to handle the changes in the amount of physical resources allocated to it in a smooth fashion. Some of the most prominent physical resources of an OS are processors, physical memory, and I/O devices.

The current tendency among the noncommodity symmetric multiprocessor (SMP) system vendors is to develop systems that are increasingly large in terms of the number of processors, number of I/O slots, and memory size. Although advances in the design of hardware continue to provide rapid increases in the

sizes of these physical resources, a number of major applications and subsystems often lag behind in scalability; hence, the trend in high-end SMPs is to support partitioning of large SMPs and to use these systems for effective server consolidation. Partitioned SMPs typically come in two kinds: systems with physical partitions (PPARs) and systems with logical partitions (LPARs). In a physically partitioned system, the granularity of partitioning is typically coarse, because the partitioning occurs at physical boundaries such as system boards. In a logically partitioned system, the granularity of partitioning is typically much more fine-grained, such as a single CPU or even a fraction of a CPU, a small block of memory, or an I/O-slot instead of an entire I/O-bus. Hence a given SMP can be subdivided into many more LPARs than PPARs.

IBM first provided LPAR support in the Advanced Interactive Executive (AIX*) operating system with the introduction of the pSeries* 690 system in December 2001. This first release of LPAR support was static in nature, that is, the reassignment of a resource from one LPAR to another LPAR cannot be made while AIX is actively running, and both the donor LPAR and the receiver LPAR must be rebooted to enable a reassignment. For such a system to provide support for various resource-related autonomic computing features, such as dynamic resource balancing across LPARs, Capacity on Demand, Dynamic CPU Sparing, and hot swapping, it needs to augment the static partitioning capabilities with dynamic LPAR (DLPAR) capabilities. As of 2002, the pSeries 690 supports the dynamic reassignment of resources across LPARs running AIX. In AIX, this functionality is referred to as dynamic reconfiguration (DR). Since AIX is an enterprise UNIX** operating system that has been designed to be robust, high in performance, rich in functions and support of platforms, and hence monolithic, the addition of a valuable autonomic computing feature such as DR has to be carefully morphed into the existing semantics, code base, and structural organization of the operating system. These challenges found in adding autonomic computing capabilities are encountered by most of the large systems with a significant installation base. Later in this paper, we briefly describe the design of DR in AIX and show that with carefully developed designs, autonomic computing capabilities can be added to an enterprise quality OS, while preserving its performance, semantics, and structural organization. Besides describing the designs within AIX that enable the smooth migration of physical resources, we also describe how these designs are being exploited to provide a variety of valuable autonomic computing features to an IT establishment.

**Autonomic benefits of DLPAR.** DLPAR in a pSeries 690-AIX system offers a great deal of flexibility to users, allowing resources to be shifted to where they are most needed without impacting system availability. The DLPAR technologies that have been developed provide the basic building blocks on which many self-optimizing, self-configuring, self-protecting, and self-healing features of the system are built. These features enable the implementation of autonomic system management and goal-oriented policies to optimize the performance and usage of system resources. DR also improves the levels of resource utilization and the reliability and serviceability (RAS) characteristics of the SMP, that translate into real cost savings for the IT establishment.

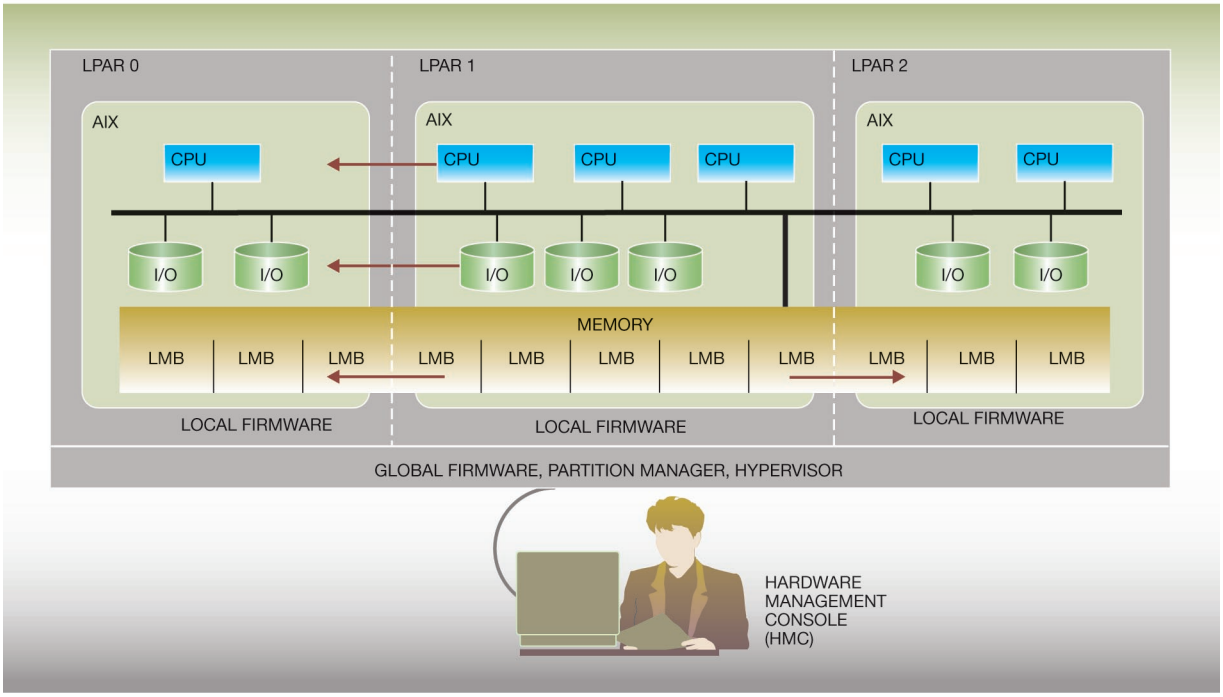
Some of the benefits offered by an SMP with LPAR capabilities are:

1. Servers can be consolidated by simply placing the workloads of several smaller servers into separate LPARs of a big SMP, hence reducing and unifying systems administration tasks.
2. Workloads can be separated by designating separate LPARs to run different workloads, for example, one LPAR for development work, one LPAR for testing, and several LPARs for production workloads.
3. The running of an application/subsystem/OS at its optimal performance and scalability can be obtained on an LPAR with optimal amounts of physical resources for that specific instance of application/subsystem/OS.

DLPAR additionally enables the following autonomic features in a system:

- Dynamic Capacity on Demand (DCOD)—DLPAR enables cross-partition workload management, which is particularly important for server consolidation, in that it can be used to better leverage system resources across partitions, thereby achieving higher levels of resource utilization, resulting in enhanced system throughput. Here is a possible usage scenario: The LPARs on an SMP are the servers for workloads originating from users in different time zones of the country, or even from different cities around the globe. While one LPAR "sleeps," its spare resources can be shifted to another LPAR that "wakes up" to do its work for the day. This shifting can be done manually via oper-

Figure 1 Logical partitioning in pSeries SMPs




ator command, and then later can be automated via the Global Resource Manager (GRM, an automated resource balancer across a specified group of LPARs in an SMP, based on OS utilization and needs) or the enterprise WorkLoad Manager (eWLM, an end-to-end response-time-based load balancer for "instrumented" applications spanning LPARs in an SMP, or even across SMPs).

- Dynamic Capacity Upgrade on Demand (DCUoD)—DLPAR enables the upcoming DCUoD feature of the pSeries 690 by allowing customers to purchase a server with extra unlicensed resource capacity, and later license and add this capacity dynamically to running AIX LPARs as their resource requirements increase.
- Dynamic CPU Guard and Dynamic CPU Sparing—DLPAR allows systems to smoothly replace processors that show intermittent, but correctable, errors. This self-healing feature will continue to become important with reduction in the silicon device size along with greater and greater integration on a chip. The Dynamic CPU Guard feature is an improved and dynamic version of the existing RAS feature named CPU Guard in earlier AIX versions. The older CPU Guard feature predicts the failure of a running CPU by monitoring certain types of transient errors and dynamically takes the CPU off line, but it does not provide a substitute CPU, so that a customer is left with less computing power. Additionally, the older feature will not allow an SMP to operate with less than two processors. The DLPAR technologies allow the OS to function even with one processor. In addition, the Dynamic CPU Sparing feature allows the transparent substitution of a good unlicensed processor for one that is suspected of being defective. This on-line switch is made seamlessly, so that applications and kernel extensions are not impacted. The new processor autonomously replaces the defective one. Dynamic CPU Guard and Dynamic CPU Sparing work together to protect a customer's investments through their self-diagnosing and self-healing software. Both features are planned to be available on pSeries 690 servers in AIX 5.2.

## The IBM pSeries DLPAR system architecture

The initial release of DR will be supported on the POWER4 pSeries 690 and 670 servers. Figure 1 illus-

trates the RS/6000* system architecture for DLPAR. In the diagram, LMB stands for logical memory block and is the granularity of physical and logical memory assigned to an LPAR. In AIX 5.2, an LMB will consist of 256 MB of contiguous memory. The size of an LMB is expected to decrease in future releases of AIX.

In this section we list some of the pSeries system components that had to be modified to become DR-aware in order to implement DLPAR. Some of these components were introduced during the implementation of static LPAR (e.g., the hardware management console, hypervisor, global firmware, and two registers for partition memory management), and some components existed even before LPAR existed (e.g., local firmware and AIX). We did not have to introduce any new components for the implementation of DLPAR; we only made changes to existing ones.

**Hardware management console.** The hardware management console (HMC) is the main control point for DLPAR configuration definitions and operations. In the initial release, these operations are controlled mainly through a new resource management graphical user interface (GUI) that runs on the HMC. This GUI invokes a new HMC command that encapsulates the DLPAR request and provides the necessary sequencing, so that the firmware and the OS can act in a coordinated fashion to achieve the desired result. Internally, this new HMC command notifies both the OS and the global firmware of a remove or add request. In time, this command will be used by other programs such as the GRM to dynamically transfer resources based on capacity requirements.

**Global and local firmware.** Global firmware provides the basic DLPAR enablement through machine-dependent logic that is designed to start, stop, and electronically isolate physical resources within the context of a logical partition. That is, global firmware performs these operations indirectly at the request of the OS, which actually passes these requests to the local firmware of the LPAR. In general, the local firmware does not contain machine-dependent logic, but is used to provide an abstraction to the OS of the logical resources that are currently assigned to the LPAR. The OS utilizes the local firmware interfaces to determine the identity and physical characteristics of logical resources that are present in the LPAR and to control them. For example, local firmware provides interfaces to start, stop, isolate, and unisolate logical resources by invoking functions provided by the global firmware.

**AIX.** When the OS receives a request to add or remove a resource, it has to take actions, both in the user space and in the kernel, to achieve the desired result. These actions are described in more detail in the next section.

## AIX DR components

Dynamic reconfiguration support in AIX 5.2 is provided for three types of hardware resources: processors, memory, and I/O slots. In this AIX release, the granularity of addition or removal for processors is one CPU; for memory, 256 MB; and for I/O, one PCI (Peripheral Component Interconnect) slot. In future releases of AIX, finer granularities are planned for both memory and processor. The following subsections briefly describe the design for the addition and removal of each hardware resource type, and the kernel modifications required for enabling DR in AIX. Although we mostly focus on DR as an enabler of autonomic computing in this paper, we have also used autonomic computing principles even within the DR design, an example of which is given in a subsequent subsection entitled "Dynamic removal of memory."

**DR of processors.** Achieving dynamic reconfiguration of processors introduced changes to the base kernel as follows:

1. For consecutive CPU identifiers (IDs), the kernel keeps track of its CPUs by assigning a distinct number, called a logical CPU ID, to each of its CPUs. Prior to DR, the kernel assumed that these logical CPU IDs were consecutive numbers, that is, no holes were allowed. To be able to randomly remove and add CPUs without perceptible disruption to applications, we had to modify the kernel so that it can tolerate missing items in the numbering of logical CPU IDs. Though we could have modified the kernel whichever way we wanted, there are many third-party applications (e.g., device drivers and performance tools) that assume consecutive numbering of CPUs, and they must be provided with binary compatibility, so as to fulfill the backward compatibility objective of AIX. Hence, another layer of numbering, bind CPU IDs, was introduced. The bind CPU ID abstraction provides a consecutive numbering of on-line CPUs for applications, even when the logical CPU IDs in the kernel are randomly removed and added, that is, the list of logical CPU IDs now becomes a list of on-line CPUs and off-line or removed CPUs.

2. For MP/UP locks, some components of the kernel were coded to decide at boot time whether they will acquire uniprocessor (UP) locks or multiprocessor (MP)-capable locks during the lifetime of the OS session. These components were modified so that they will function properly even when DR changes the system dynamically from a UP to an MP (and vice versa).

*Dynamic removal of processors.* Dynamic removal of a processor involves the following tasks initiated from the OS:

1. Notify DR-registered applications and kernel extensions, if any, so that they will voluntarily remove dependencies on the CPU to be removed. This task typically involves unbinding the threads that are bound to the CPU being removed.
2. Migrate threads bound to this CPU to another running CPU.
3. Retarget pending interrupts, and change the bindings of all interrupts currently bound to the processor to be removed. This task involves changing the interrupt controller data structures.
4. Migrate the timers and threads from the CPU being removed to another CPU within the same LPAR.
5. Notify the hypervisor or firmware to complete the removal task.

*Dynamic addition of processors.* Dynamic addition of a processor involves the following tasks initiated from the OS:

1. Create a process (waitproc) for idle looping on the incoming CPU before it starts to do real work.
2. Set up various hardware registers (e.g., SDR1 = Search Descriptor Register 1, which defines the start physical address and size of the page table in memory, GPR1 (General Purpose Register 1) for kernel stack, GPR2 for the kernel table of contents, etc.) of the incoming CPU.
3. Allocate or initialize, or both, the processor-specific kernel data structures (dispatcher run-queue, per-processor data area, interrupt stack, etc.) for the incoming CPU.
4. Add support for the incoming CPU to the interrupt subsystem.
5. Notify the DR-registered applications and kernel extensions that a new CPU has been added.

**DR of memory.** The implementation of dynamic reconfiguration of memory in AIX 5.2 enables the removal and addition of 256 MB contiguous sections of memory. This unit is referred to as a logical memory block (LMB). A smaller-sized LMB, for example, 16 MB, will be allowed in future releases of AIX.

Two important challenges were resolved during the implementation of memory DR: In the first one, the physical addresses of some memory are exposed to manipulation by applications over which the kernel does not have direct control. These accesses are allowed for functional reasons in some cases (e.g., direct memory access, or DMA), and for performance reasons in other cases (e.g., pretranslated addresses). Pretranslated addressing is an internal feature of AIX with which virtual-to-physical address translations for a data buffer to be involved in a DMA operation is done only once for the life of the data buffer. DR of page-frames with these uses can be handled in at least two ways: (1) by modifying the kernel and firmware so that such accesses by kernel extensions to the page-frame being removed can be controlled; or (2) by requiring kernel subsystems to register a DR callback function with the kernel DR subsystem, and invoking the callback function at memory removal time.

We have applied both techniques, choosing one over the other, depending on the specific circumstances, while minimizing the impact on system performance as well as kernel changes. In the case of DMA, the kernel and firmware control the access to the memory being removed by selectively disabling the bus traffic while the contents of the memory with DMA are being migrated to a new location. In the case of pretranslated addresses, a callback mechanism is used.

The second challenge is enabling the translation-on execution of the majority of the kernel, which previously was run in translation-off mode and hence required maximally sized data structures to be allocated at boot time for the maximum amount of physical memory that the AIX/LPAR instance can potentially grow into.

*Dynamic removal of memory.* For the purpose of dynamic removal, we classify the memory page frames of AIX into five categories: unused, pageable, pinned, DMA-mapped, and translation-off memory. The approach taken to remove a page-frame (4096 bytes) in each of these categories is as follows:

1. A page-frame containing a free page is simply removed from its free-list.

2. A page-frame containing a pageable page can be made to either page out its contents to disk or to migrate its contents to a different free page-frame. In the future, an autonomous agent, for example, GRM or eWLM, can choose one of these two approaches, depending on its knowledge of memory availability.

3. A page-frame containing a pinned page will have its contents migrated to a different page-frame; also the page-fault reload handler had to be made to spin during the migration.

4. A page-frame containing a DMA-mapped page cannot be removed or have its contents migrated until all the accesses to the page are blocked. Here, the term "DMA-mapped page-frame" is used generically to mean a page-frame whose physical address is subject to read or write by an external (to kernel) entity such as a DMA engine. The contents of a DMA-mapped page-frame are migrated to a different page-frame with a new hypervisor call (*h_migrate_dma*) that will selectively suspend the bus traffic while it is modifying the TCEs (translation control entries) in system memory used by the bus unit controller for DMA accesses.

   Other page-frames whose physical addresses are exposed to external entities are handled by invoking preregistered DR callback routines and then waiting for completion of the removal of their dependencies on the page.

5. A page-frame containing translation-off pages will not be removed by DR in AIX 5.2. Fortunately, there is just a small amount of these page-frames, and they are usually colocated in low memory. These page frames are intended to be handled in later AIX releases.

The design and implementation of dynamic memory removal has manifested itself as three modular functions, such that one can mix and match these functions in several possible ways, adapting to the state of the system at the time of memory removal, thus achieving the desired end result with the most optimal path. This design adheres to the self-optimizing principles of autonomic computing, as described in Reference 2. These three modular functions perform the following three tasks respectively on the memory (LMB) being removed: (a) remove its free and clean pages from the regular use of VMM (Virtual Memory Manager), (b) page-out its pageable dirty pages, and (c) migrate the contents of each remaining page-frame in the LMB to a free page-frame outside the LMB. Memory removal can be implemented with any one of the sequences: abc, ac, bc, or just c.

For example, the decision to either invoke page-out (task b) or to migrate (task c) all the pages depends on the load in the LPAR at that particular time. If the LMB being removed contains a lot of dirty pages that belong to highly active threads, then it does not make sense to invoke task b, because these pages will be paged back in almost immediately, negatively impacting the efficiency of the system.

As a second example, if there are not enough free frames in other LMBs to migrate the pages to, then the memory removal procedure can invoke task b before invoking task c, so that there will be far fewer pages left that need to be migrated in task c.

***Dynamic addition of memory.*** When a memory-add request arrives at AIX, it has to perform two tasks: allocate and initialize software page descriptors that will hold meta-data for the incoming memory, and distribute the incoming memory among several free-frame pools so as to preserve the behavior of memory management algorithms. The challenges encountered in implementing these two tasks and how they were resolved are now described.

The primary challenge was the allocation of memory for the software page descriptors for the incoming memory. The problem was that, prior to DR, these page descriptors could be accessed in translation-off mode while trying to reload a page mapping into the hardware page table. If the page descriptors are allowed to be accessed in translation-off mode, the memory allocated for those new descriptors has to be physically contiguous with the memory for existing descriptors, which implies that memory has to be reserved at boot time for descriptors for the maximal amount of memory that the OS instance can potentially grow into. This can potentially incur inefficiency and wastage of much memory, particularly if not utilized. We avoided this wastage by changing the kernel so that software page descriptor data structures are always accessed in translation-on mode.

Another challenge that was resolved while implementing dynamic memory addition was the difficulty in distributing the incoming memory across different page replacement daemons, so that each dae-

mon handles a roughly equal load. In AIX, memory is hierarchically represented by the data structures vmpool, mempool, and frameset. A vmpool represents an affinity domain of memory. A vmpool is divided into multiple mempools, each mempool being managed by a single page replacement least recently used (LRU) daemon. Each mempool is further subdivided into one or more framesets that contain the free-frame lists, so as to improve the scalability of free-frame allocators. When new memory is added to AIX, the vmpool that it should belong to is defined by the physical placement of the memory chip. Within that vmpool, we want to distribute the memory across all the available mempools to balance the load on page replacement daemons. However, the kernel assumed that a mempool consisted of physically contiguous memory. Thus, to be able to break up the new memory (LMB) into several parts and distribute them across different mempools, the kernel was modified to allow mempools to be made up of discontiguous sections of memory.

**DR of I/O slots.** The methods to dynamically configure or unconfigure a device have been introduced as early as AIX version 3. The changes required in the kernel design for DR of I/O slots were not in the same scale as those for DR of processors, and particularly those for DR of memory. The reason is that the onus of configuring or unconfiguring a device lies with the device driver software, which operates in the kernel extension environment. The kernel just acts as a provider of serialization mechanisms for devices accessing common resources and as an intermediary between the applications and the device drivers.

## Challenges of adding autonomic features to a mature UNIX OS

The AIX kernel is an industrial strength pageable and pre-emptable kernel that offers high performance and scalability (up to 32-way SMP) and supports many vendor device drivers, databases, and applications. The same kernel source code supports all reasonable combinations of 32-bit or 64-bit kernels, 32- or 64-bit applications, and 32- or 64-bit RISC (reduced instruction-set computer) systems with old and new RISC architectures, uniprocessors, and multiprocessors. Being friendly to its users, AIX goes out of its way to offer backward binary compatibility in a wide variety of situations. Being robust and having high performance, the critical kernel sections are skillfully guarded by a carefully crafted hierarchy of data and code locks. Implementing DR involved careful changes to numerous critical components of the AIX kernel.

## Self-healing and self-protecting features

DR also serves as the foundation for a new advanced self-healing and self-protecting technology, especially when it is coupled with the presence of extra unlicensed capacity. This new technology enhances a preexisting self-diagnosing technology, called the CPU Guard feature, that monitors recoverable error rates for processors. At its simplest level, this new technology substitutes a spare processor in a transparent fashion for a processor that the system has internally diagnosed as being defective. Spare resources are only present if the system was shipped with extra unlicensed capacity as defined by the Capacity Upgrade on Demand solution, although it should be noted that there are no license keys that have to be entered to enable this new Dynamic CPU Sparing technology. This Dynamic CPU Sparing feature does not address the case in which a CPU fails so suddenly that a state-save and an orderly live swap of the CPU cannot be performed.

The technical aspects of this new technology are outlined next. Firmware monitors the health of each processor in terms of the number of recoverable errors. If this number exceeds an internal threshold, it raises a repeat guard error to the operating system and makes available an unlicensed processor, assuming that one is available. AIX acts on this error notification by invoking the DR Manager, which guides the self-protecting procedure from the perspective of an operating system. If an unlicensed processor is not available, it proceeds to take the defective processor off line in a fashion similar to that of the existing CPU Guard feature; this procedure constitutes a self-protecting feature of the system. If an unlicensed processor is available, the procedure uses it as a substitute for the defective one, making the switch transparently with the new DR technologies. This action constitutes a self-healing feature of the system.

The DR Manager determines whether an unlicensed processor is available through the use of new firmware routines, and if it finds one, attempts to take ownership of it from the firmware by invoking new firmware routines. These new firmware routines are the same ones that are used for DR processor addition, although a different return code is used to indicate that they are reserved for self-healing. Next, the DR Manager queries the kernel to determine the

identity of the defective processor, which was named by the repeat guard error log entry. Finally, it invokes the kernel to perform the live swap.

The new processor is always started before the defective one is stopped, so that this technology can be applied to a single-processor LPAR—an improvement over the old CPU Guard self-protecting technology, which could not remove the last two on-line processors. At a high level, the switch is made by taking over the execution of the defective processor and using it to control the sequence of events that are required to transparently complete the switch, much of which has to be run on the defective processor itself, since the new processor is going to assume the logical identity of the failing one eventually. This is largely a matter of preserving the physical state of the defective processor and atomically reprogramming any funneled hardware interrupts that may be directed to the defective physical processor before allowing the new processor to begin operating.

The kernel actually performs the switch by masking external interrupts on the defective processor, so that it is not expected to respond to external events that may be generated by adapters or other processors in the partition, and by tightly controlling the execution of the new processor. Before starting the new processor, the defective processor saves the state of its registers, so that they can be restored by the new processor once it is functioning. Next, it vacates its logical CPU ID inside the kernel so that the new processor can be brought up in the proper logical CPU ID relative to the kernel. This is important in that logical workloads assigned to the defective logical processor (e.g., its threads and timers) do not need to be migrated, which in large part constitutes the transparency of the switch. There are many logical processor states and very few physical processor states that need to be taken care of. Next, it invokes firmware to start the new processor at a startup routine that is specific to this algorithm, and it waits for the new processor to indicate that it has successfully added itself to the global processor interrupt queue. Once this occurs, the defective processor can stop itself without fear of losing any external interrupts. This synchronization point is required to ensure that the global processor interrupt queue has at least one processor at all times. At this point, the new processor loads the register state that was previously preserved and resumes the execution path previously established by the defective processor.

This self-healing technology is built into the AIX base operating system and is automatically enabled by default, although the customer may disable it through system management options. This new technology is initially available on the pSeries model 690.

### Self-optimizing and self-configuring features

The next release of AIX will have provisions for the automatic movement of its logical resources between LPARs, allowing the overall utilization of the physical resources of an SMP to be increased at no extra cost to the installation. For example, an agent, such as a Global Resource Manager (GRM), will monitor the utilizations, needs, and Service Level Agreements (SLAs) of a pool of LPARs and autonomously move the DR resources from an underutilized LPAR to a needy one, hence enriching the SMP with self-optimizing and self-configuring capabilities in a timely fashion. GRM also ensures that the DR movements will be orchestrated in a smooth fashion and without undesirable oscillations.

Additionally, as part of the IBM autonomic initiative, work is ongoing to provide optimal end-to-end response time for "instrumented" Web and commercial applications that span the LPARs of an SMP, as well as applications that span SMPs with or without LPARs, based on SLAs.

## Conclusion

The DLPAR and DR technologies that have been developed on the IBM pSeries 690 servers have enabled these servers to become truly autonomic computer servers. As described in this paper, these servers have features that are self-protecting and self-healing, and they have the basic building blocks that enable these systems to be self-configuring and self-optimizing. The self-configuring and self-optimizing features are currently planned to be available in the near future. Thus, all four self-managing features that are at the heart of an autonomic server will soon be available on the pSeries 690. DLPAR is a strategic pSeries AIX capability and will also be made available on future pSeries servers.

In a possible future enhancement, virtualization of a physical processor into virtual processors allows the processor resource to be sharable in a fine-grained fashion (instead of one processor at a time) among a pool of LPARs in the SMP.

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of The Open Group.

## Cited references


1. P. Horn, *Autonomic Computing: IBM's Perspective on the State of Information Technology*, IBM Corporation (October 15, 2001); available at http://www.research.ibm.com/autonomic/manifesto/autonomic_computing.pdf.
2. K. Ekanadham et al., *Anatomy of Autonomic Server Components*, Research Report RC 22637, IBM T. J. Watson Research Center, Yorktown Heights, NY.



*Accepted for publication August 16, 2002.*



**Joefon Jann** *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (electronic mail: joefon@us.ibm.com).*

Ms. Jann is a Senior Technical Staff Member at the Watson Research Center, where she leads a small team that conceived and prototyped the notion of DLPAR for AIX. She is currently working on automating DLPAR. Her previous projects in IBM include the design and prototype of a DSM (distributed shared memory) system for the pSeries servers, the design and implementation of the LoadLeveler® hierarchical communication infrastructure, which enables IBM's LoadLeveler product to work in the largest SP™ installations. She coinvented the "Jann MPP Workload Model," was a member of the Deep Blue Computer chess team, a developer of the IBM product VMPRF (VM Performance Reporting Facility), a VM/SNA Area Specialist, and an APL programmer. Ms. Jann was a lecturer in mathematics at Lehman College for three years, and holds B.A. and M.A. degrees in pure mathematics from Wellesley College–MIT and the City University of New York (CUNY), respectively, and an M.S. degree in computer science from Columbia University. She is a member of the IEEE.


**Luke M. Browning** *IBM Server Group, 11501 Burnet Road, Austin, Texas 78758 (electronic mail: browninl@us.ibm.com).*

Mr. Browning is a Senior Technical Staff Member in the AIX Kernel Architecture and Design Department of the IBM Server Group, working on dynamic LPAR, workload management, threads, and process management. He joined IBM in 1984 in Austin after receiving his bachelor of science degree in computer science from the University of Texas at Austin.


**R. Sarma Burugula** *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (electronic mail: burugula@us.ibm.com).*

Mr. Burugula is an advisory software engineer at the Watson Research Center. He received his B.S. degree from Regional Engineering College Warangal and an M.S. degree from the Indian Institute of Technology Kanpur in India, both in computer science. He joined IBM in 1996 and has been working primarily on the IBM pSeries platform, developing various parallel and scalable subsystems.

# Text analysis and knowledge mining system

by T. Nasukawa
T. Nagano

***Large text databases potentially contain a great wealth of knowledge. However, text represents factual information (and information about the author's communicative intentions) in a complex, rich, and opaque manner. Consequently, unlike numerical and fixed field data, it cannot be analyzed by standard statistical data mining methods. Relying on human analysis results in either huge workloads or the analysis of only a tiny fraction of the database. We are working on text mining technology to extract knowledge from very large amounts of textual data. Unlike information retrieval technology that allows a user to select documents that meet the user's requirements and interests, or document clustering technology that organizes documents, we focus on finding valuable patterns and rules in text that indicate trends and significant features about specific topics. By applying our prototype system named TAKMI (Text Analysis and Knowledge MIning) to textual databases in PC help centers, we can automatically detect product failures; determine issues that have led to rapid increases in the number of calls and their underlying reasons; and analyze help center productivity and changes in customers' behavior involving a particular product, without reading any of the text. We have verified that our framework is also effective for other data such as patent documents.***

Since a textual format is a very flexible way to describe and store various types of information, large amounts of information are stored and distributed as text. Moreover, the amount of accessible textual data has been increasing rapidly. Such data may potentially contain a great wealth of knowledge. However, analyzing huge amounts of textual data requires a tremendous amount of work in reading all of the text and organizing the content. Thus, the increase in accessible textual data has caused an information flood in spite of hope of becoming knowledgeable about various topics.

In order to overcome this counterintuitive situation, a text mining technology called TAKMI (Text Analysis and Knowledge MIning) has been developed to acquire useful knowledge from large amounts of textual data such as internal reports, various technical documents, messages from various individuals, and so on. It is described in this paper. In particular, we try to analyze what a large set of documents indicates as a whole rather than focus on the specific information in each document.

The most important issue for this text mining technology is how to represent the contents of textual data in order to apply statistical analysis. Since any information that cannot be retained within the representation cannot appear in the final output, we should carefully examine the expressive power of the representation system. Then, once the appropriate information is extracted from the text, the next issue is how to provide statistical analysis. We should then apply appropriate mining functions adapted to the representations of the original content of the text. Finally, since the content of the text varies greatly, it is essential to visualize the results and allow an interactive analysis to meet the requirements of analysts working from multiple points of view.

Table 1 Comparison of document-handling technologies

| Function | Purpose | Technology | Data Representation | Natural Language Processing | Output |
|---|---|---|---|---|---|
| Search documents | Focus on data related to some specific topics | Information retrieval | Character strings, keywords | Keyword extraction (conversion to base forms) | A set of documents |
| Organize documents | Overview of topics | Clustering,[1] classification[2] | Set of keywords (Vector Space Model)[3] | Analysis of keyword distribution | Sets (clusters) of documents |
| Discover knowledge | Extract interesting information from content | NLP, data mining, visualization | Semantic concepts | Semantic analysis, intention analysis | Digested information (trend patterns, association rules, etc.) |

In the next section of this paper, we contrast our text mining technology against other document-handling technologies, such as document search and document classification, as well as other work in text mining. In the third section, we describe the technical details of our technology, especially focusing on natural language processing to extract concepts from text, and we also describe our data mining technology to find interesting patterns and features, as well as our visualization technology that allows interactive analysis. The fourth section is devoted to the successful application of our text mining technology to records in customer contact centers such as PC help centers. The expressive power of the representation system for concept extraction is evaluated by using the customer contact records in this section. Applications for other data such as patent documents are discussed in the fifth section, followed by concluding remarks.

## Text mining technology

We first describe various document-handling technologies and related work in text mining technology.

**Document-handling technologies.** In order to reduce the work in handling huge amounts of textual data, various technologies have been developed, and the features of some document-handling technologies are classified in Table 1.[1–3]

Information retrieval technology is probably the most common technology to use when we are faced with a very large number of documents. The term "text mining" (or "text data mining") is sometimes used to indicate this technology because it detects and extracts documents that we want from mountains of documents, and it allows us to select data related to some specific topics that we are interested in so that the amount of data we have to handle is reduced without losing the information we want. However, we have to specify what we want in the form of a query in order to use this technology. Thus, the use of this technology is limited when we do not have a clear intention about what to search for and knowledge of what can be retrieved from the database we are searching. Moreover, even when we have some specific topics to search for and successfully make some queries, the output we obtain is a list of documents that we still have to read to find the information, unless we are simply interested in such data as the number of documents that contain specific keywords or character strings.

Document organization technology can give us an overview of a document archive, either by classifying documents into predefined classes or by clustering documents with similar contents, where similarity is defined by the system. By using this technology, we can find out what kind of topics are contained in the document archive and the sizes of the classes or clusters for each topic. The term "text mining" is also used to indicate this technology, because it may break up and organize mountains of documents. Yet we still have to read each document in the class or the cluster that we want to analyze for details.

In contrast, "text mining" in the sense of knowledge discovery can be characterized as a novel approach to the data overload problem that the search and organization paradigms attempt to alleviate as in Hearst.[4] In our work, we are aiming at a technology to analyze more detailed information in the content of each document and to extract interesting information that can be provided only by multiple documents viewed as a whole, such as trends and sig-

nificant features that may be a trigger to useful actions and decision-making. For example, in a large number of documents related to customers' calls (as shown in later examples) we want to find what kinds of topics have recently been increasingly mentioned and which product is associated with specific topics, so that we can take appropriate actions for the improvement of call center productivity and product quality.

**Related work in text mining technology.** In terms of knowledge extraction, many kinds of knowledge can be extracted from textual data, such as linguistic knowledge for Natural Language Processing (NLP)[5] and domain-specific lexical and semantic information that may be stored in a database.[6] The technology of Information Extraction[7] is also related in terms of extracting meaningful items from textual data. However, Information Extraction, typically focused in Message Understanding Conferences (MUCs),[8] is intended to find a specified class of events, such as company mergers, and to fill in a template for each instance of such an event. Thus, this technology is almost the inverse of our text mining that aims to find novel patterns rather than predefined patterns in a specified class. In our research, we focus on extracting information that may be used to discover trends in the domain described in the textual database. Consequently, we are not concerned with providing analyses of documents but rather of a subset of the textual database viewed as a whole. At the same time, we focus on technologies that may also be used to discover relationships between separate categories. Thus, our "text mining" is a text version of generalized data mining, and it consists of NLP to extract concepts from each piece of text, statistical analysis to find interesting patterns among the concepts, and visualization to allow interactive analysis.

In spite of strong demand from individuals who deal with large amounts of text, there has been little practical work in applying data mining techniques to text. Among the small number of publications on using data mining technology for text, Feldman and Dagan[9] presented a framework for finding interesting patterns in the distributions of concepts in documents, and Feldman et al.[10] introduced a method for visualizing associations of concepts in a certain context. Lent et al.[11] presented a technique for finding trends in the use of words and phrases in documents. However, the output of these previous methods is limited by the shallow NLP. Each document is represented by a simple character string or a set of keywords as in Mladenic.[12] When keywords are extracted from each document, linguistic knowledge may be applied to convert conjugated forms and plural forms into canonical forms and singular forms so that the number of variants of the keywords is reduced. Synonyms may also be unified into a canonical expression. However, a lot of information is conveyed with functional words, such as negations. Such words are usually treated as stop words, which are ignored. Also, the relationships among words are lost. Moreover, these technologies do not discriminate between information conveyed with higher-level sentential structures, such as the interrogative mood and the imperative mood. Such information is crucial to recognize communicative intentions such as questions, requests, complaints, and commendations, which is especially important to make a useful analysis of data related to customer relationship management (CRM) as described in a later section. Thus, we developed a framework for extracting concepts that consist of predicate argument pairs in association with information on modality, which often indicates intention and polarity, either negative or positive, which is also an important factor in analyzing intentions.[13]

## Framework of text mining

In this section, we discuss issues of concern for obtaining useful results in text mining, followed by our approach to the development of a practical text mining system, since these issues affect each of the following three components that constitute the framework.

1. Concept extraction based on robust natural language processing
2. Data mining for discovering rules and patterns
3. Visualization and interactive analysis

**Concept extraction for text mining.** In this paper, we use the term "concept" as a representation of the textual content in order to distinguish it from a simple keyword with the surface expression.

*Issues in representation of textual contents.* The first problem is that, because of the ambiguities in natural language, the same keyword may express entirely different meanings. For example, the word "Washington" may represent a person, place, or something else. The meaning of such polysemous words is normally determined according to their context.

The inverse problem is that different expressions may refer to the same meaning, for instance, "car" and "automobile" or "H/W" and "hardware." Even when the meaning may not be exactly the same, it may be necessary to treat these expressions as denoting the same meaning for text mining, especially when some of the synonyms are used infrequently, in order to avoid data sparseness, since a small number of appearances compared to others tend to be ignored in the final output.

Thus, the surface expression of keywords is not a proper representation for text mining. And the representation of context should have the capability of reflecting the results of semantic disambiguation based on context analysis in order to deal with the polysemy and the synonymity.

Moreover, insertion of a single word such as "not" or alteration of word position may change the entire meaning of a sentence as in the following examples.

(a) X did fail.
(b) X did not fail.
(c) Did X fail?

However, in the bag-of-words approach, these sentences are treated as the same content since they share the same set of keywords. Besides the negation and interrogative mood, some auxiliary verbs such as "can" and some verbs such as "want" often indicate the author's communicative intentions.

Furthermore, the relationships among words in a document are only defined as co-occurrences in the bag-of-words approach, which, for example, may lead us to a misinterpretation of a relationship between "B" and "good" in the following sentence.

A is good, but B is bad.

According to Matsuzawa,[14] only 40 percent of the predicate and noun pairs within the same sentence have grammatical dependencies. Since such grammatical dependencies may convey useful meaning such as a description of the actions of the subject, the representation of the dependencies is important. In addition, the role of modifee or modifier, such as being subject or object, is also important, as in the following example.

(a) A deleted B.
(b) B deleted A.

*Concept extraction in TAKMI.* In order to deal with the above problems, we took the following approach to represent the content in textual data for our prototype text mining system named TAKMI.

- Assign semantic features to words and phrases expressed in the text and unify synonyms into a canonical form by referring to a semantic dictionary, in order to categorize them semantically so that they are treated as unique concepts.

  Washington [person]
  Washington [place]
  fail [complaint][15]

- Associate communicative intentions with predicates by analyzing grammatical features and lexical information so that "fail" in the previous example (a) to (c) may be represented as follows:

  (a) X did fail. → fail[complaint]
  (b) X did not fail. → not fail[commendation]
  (c) Did X fail? → fail[question]

  instead of

  X and fail

  in the bag-of-words representation.

- Extract dependency pairs of words and phrases so that the content of the previous examples may be represented as follows:

  Program A is good, but Program B is bad.
  → Program A[software] . . . good[commendation],
  Program B[software] . . . bad[complaint]

  Program A deleted Program B.
  → Program A[software] . . . delete[complaint],
  delete[complaint] . . . Program B[software]

In order to generate these representations, in TAKMI we use the steps described in the following subsections.

Creation of semantic dictionary. Assuming that each domain has important terms for analysis, we make a list of words extracted from the textual database sorted by their frequency and ask domain experts to assign semantic categories to words and phrases that they consider important, as well as to assign the appropriate canonical forms to take care of synony-

mous expressions or variations in the expressions. This dictionary consists of entries with surface representations, parts of speech (POS), canonical representations, and semantic categories such as the following example for a PC help center.

| | | | |
|---|---|---|---|
| batt | noun | battery | hardware |
| bty | noun | battery | hardware |
| card driver | noun | PCMCIA driver | software |

Since the word distribution is not balanced (as we will describe it in a later section), the number of frequently appearing words is relatively limited in a textual database, especially when its content belongs to a narrow domain. The workload for this dictionary creation has been relatively small in our experience (as also described in a later section).

Intention analysis. Intention analysis is performed by matching patterns of grammatical forms or certain expressions and by searching in a semantic dictionary in the following manner.

1. Use a POS tagger to assign a POS and a base form to each word.
2. Assign semantic features to each word and phrase by looking it up in the semantic dictionary.
3. Cluster words into verb groups and noun groups by using a shallow parser.
4. Assign intentions to verb groups by matching POS patterns and their base forms using heuristic rules such as

   want to + VERB → VERB[request]
   please + VERB → VERB[request]

Dependency analysis. Dependency analysis is basically done by grammatical analysis but with the aim of focusing on important issues concerning the text mining application. The specification of the semantic category of words, including an indication of the intention, is effective. For example, to facilitate the analysis of problems in software, extraction of dependency pairs of a predicate indicative of problems and a noun with a semantic category [software] is generally a robust method. Thus, we apply dependency analysis to the results of intention analysis. The basic procedure of the dependency analysis is to check the local grammatical dependencies among verb groups and noun groups clustered in the intention analysis, and extract predicate and argument pairs such as a subjective noun and its predicative verb.

In order to sustain the robustness of the analysis, a noun group and a verb group within a sentence without a verb group or conjunctions in between may be considered to have a dependency.

**Data mining functions for text mining.** Once appropriate concepts are extracted from each piece of text, we can apply various statistical analysis methods in data mining to the set of concepts as well as to structured data. As a result, even a simple function that examines the increase and decrease of occurrences of each concept in a certain period may allow us to analyze trends in topics. Moreover, the semantic classification of concepts enables us to analyze the content of texts from the viewpoints of various semantic categories. An example of this function is shown later in Figure 2.

The semantic classification also enables us to find significant features for some topics by analyzing associations among concepts. Although finding association rules[16] is one of the most attractive functions in data mining in terms of discovering novel facts, analysis of association rules in very large text samples does not provide much useful information if we simply treat each piece of text as a set of items consisting of content words within the text. Because of the large number of variations in items (words) extracted from text, the number of association rules is usually too large to verify. In addition, most of them are elements of compound words that naturally occur together. However, by using the semantic categories, we can detect the significant associations based on the assumption that concepts in the same semantic category have a similar distribution of associations in comparison to other concepts.

Table 2 is an example of the highlighting of significant associations among the concepts of [liquid]s and [problem]s in the data of a PC help center. In this table, described as a two-dimensional map in TAKMI, each cell represents an association rule with a "support" number that indicates the number of textual records that contain both of the concepts in the leftmost cell and the top cell, and a "confidence" value within parentheses that indicates the percentage of the support number divided by the number of textual records that contain the concept in the leftmost cell. Those concepts are listed in order, according to the number of texts that contain them. This table tells us that "soda" in [liquid] is strongly associated with "sticking" in [problem]. The numbers in the cell, "12 (12.63%)," mean that there were 12 calls from customers who mentioned both "sticking"

Table 2 Analysis of association among [liquid]s and [problem]s

| | Damage | Fail | Sticking | Dead | Bad | Freeze |
|---|---|---|---|---|---|---|
| **Water** | 94 (11.1%) | 27 (3.19%) | 5 (0.59%) | 21 (2.48%) | 17 (2.01%) | 16 (1.89%) |
| **Coffee** | 31 (6.87%) | 12 (2.66%) | 13 (2.88%) | 7 (1.55%) | 6 (1.33%) | 5 (1.11%) |
| **Juice** | 3 (2.94%) | 1 (0.98%) | 7 (6.86%) | 4 (3.92%) | 5 (4.9%) | 0 (0.0%) |
| **Soda** | 7 (7.37%) | 2 (2.11%) | 12 (12.63%) | 4 (4.21%) | 1 (1.05%) | 1 (1.05%) |
| **Tea** | 3 (7.5%) | 1 (2.5%) | 1 (2.5%) | 0 (0.0%) | 0 (0.0%) | 1 (2.5%) |
| **Beer** | 2 (5.88%) | 0 (0.0%) | 0 (0.0%) | 0 (0.0%) | 0 (0.0%) | 4 (11.76%) |

and "soda," and those 12 calls occupy 12.63 percent of the calls mentioning "soda." Since this percentage, 12.63, is much higher than percentages for other items in [liquid], the cell is automatically highlighted.

In addition, a "topic extraction" function based on the work of Nomiyama[17] allows us to spot abnormal trend situations by analyzing the number of records in certain categories compared to variations in the background level of activity. This approach analyzes changes of topics by detecting concepts whose frequency has suddenly increased during a specific period, and it arranges the concepts in the order of their frequencies at the peak and by considering the duration of the peak. Real-world examples of this function are provided later in connection with Figures 7 and 11.

**Visualization and interactive analysis for text mining.** In order to deal with requirements from various viewpoints and to compensate for the low accuracy of linguistic analysis, it is indispensable to provide functions to visualize the results for intuitive understanding and to allow interactive analysis. We are using Information Outlining technology[18] for this purpose. Information Outlining enables a user to obtain an overview of the features and trends of a set of data by showing the distributions of items associated with the target set of data from various viewpoints indicated by categories, and by allowing the user to select or specify concepts to narrow or broaden the target sets of data. Mining functions in the previous subsection can be invoked from this visualization component for interactive analysis.

The GUI (graphical user interface) of TAKMI is shown in Figure 1. It consists of four main frames as follows:

- Frame A shows the number of records of the current analysis associated with the search criteria to focus on the set of texts. In this example, we are analyzing 12 883 records that contain "Windows** 98."
- Titles of the texts are shown in Frame B, and the actual content of each text is displayed by clicking the title such as "CDROM" or "PLANR" in this frame.
- The distribution of concepts associated with the current set of texts is shown in Frame C in accordance with the categories of the semantic features or intentions. In this frame, concepts can be sorted according to their absolute frequency, relative frequency within the data, or alphabetical order of concepts. The relative frequency is calculated using the following formula.

$$\text{Relative frequency} = \frac{\dfrac{\text{(Number of records referring to the concept among the selected records)}}{\text{(Number of records selected)}}}{\dfrac{\text{(Number of records in the complete set of records referring to the concept)}}{\text{(Total number of records)}}}$$

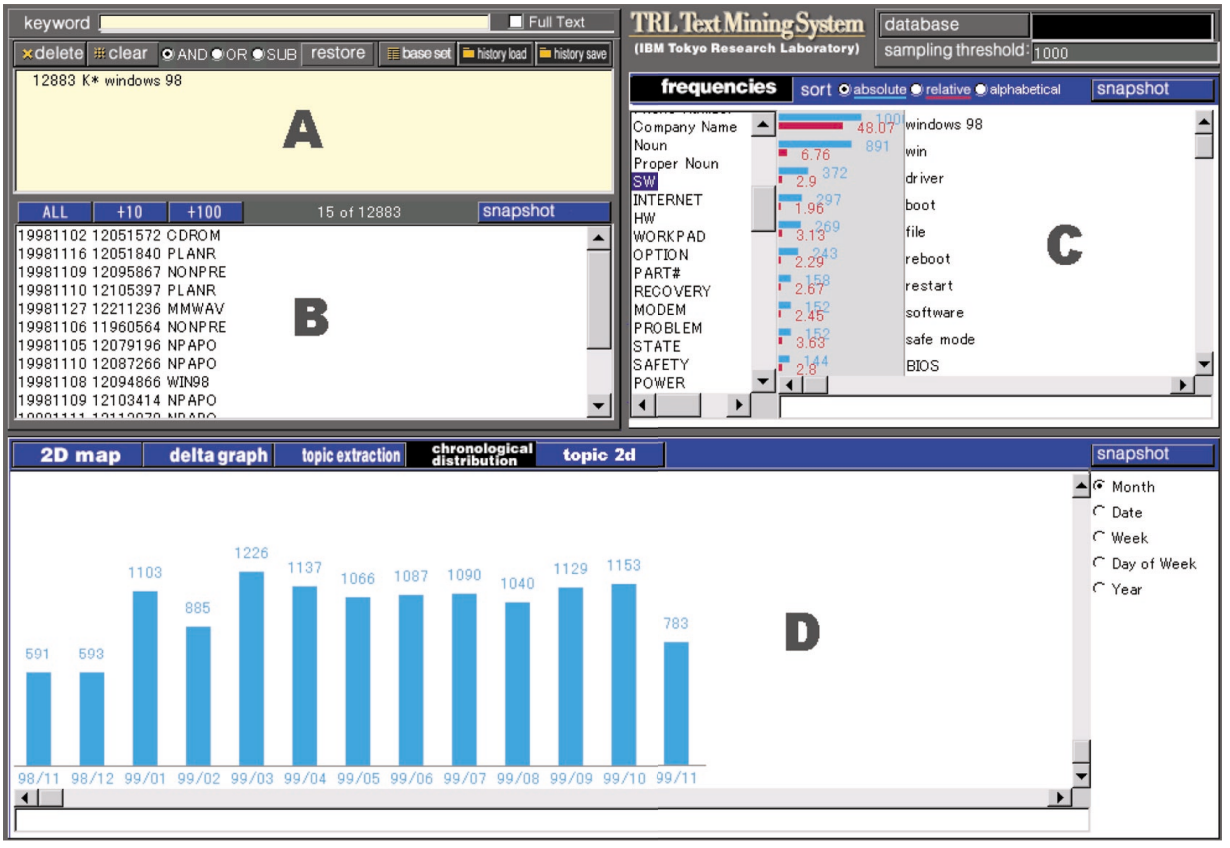
  Thus, a concept with a higher relative frequency value may be strongly related to the current data set.

- The results of the mining functions are shown in Frame D.

Thus, by using TAKMI, analysts can:

1. Detect trends such as topics whose importance is increasing and find significant associations among concepts in Frame D
2. Analyze details of the trends or significant associations by browsing the relevant concepts in Frame C

Figure 1 GUI of TAKMI




3. See the basis of these results in the original texts accessed from Frame B

## Use of TAKMI to analyze records in customer contact centers

Many companies support their customers by phone, e-mail, fax, and so on. Data on customer contacts from such support centers are a good resource for obtaining knowledge about the customers, as well as knowledge about products and services that they have purchased and used.

Each contact record generally contains both structured and unstructured data. The fixed field, structured data might contain:

- Customer information (ID [identifier], name, phone number, etc.)
- Related products or services
- Contact type (question, request, query, etc.)

However, since there is a great variety of contact types and the content of each contact is complex, it is difficult to represent each interaction precisely with a limited set of predefined categories. Consequently, a free text field is used to capture details of each contact in natural language.

Although statistical analysis of the structured data provides valuable insights that help to improve business,[19] most of the unstructured data have been ignored because analysis by hand is prohibitively expensive. For example, one of the customer support centers that we dealt with was a PC help center that receives over 50000 calls each month. A number of analysts handle these calls in the help center, and they make reports every week by reading the customer contact records. However, they make the reports based on reading about 200 to 300 records out of tens of thousands of records. Therefore, they are using less than 5 percent of all the data they have.

Thus, analysis of reports on customer contacts was a good application for text mining, since it allows inspection of all of the data possible. The source data obviously contains useful information to detect product failures, to detect issues that have led to rapid increases in the number of customer contacts and the reasons for those increases, to analyze the productivity of help desk workers, and to analyze changes in customers' behaviors toward particular products over a period of time.

We have analyzed the records of customer contact information in a PC help center in order to apply our technology and noticed the following features:

(a) Informal writing—The records have many indicators of informal writing:

- Various expressions are used for the same meaning, and there is a lack of consistency in expressions. Examples are:

  customer = cu = cus = cust = end user = user = eu

  Windows 95 = w95 = Win95

- Many sentences are ungrammatical.
- Many spelling mistakes are included.

(b) Various content types—There are several important types of contents that should be recognized and handled differently in some sentences, including: requests, questions, complaints (trouble or problem), and commendations.

(c) Multiple topics—One text may contain multiple problems and topics.

(d) Importance levels—There are several levels of importance for the concepts. For example, [safety issues] such as smoke, spark, and injury are critical for the help center.

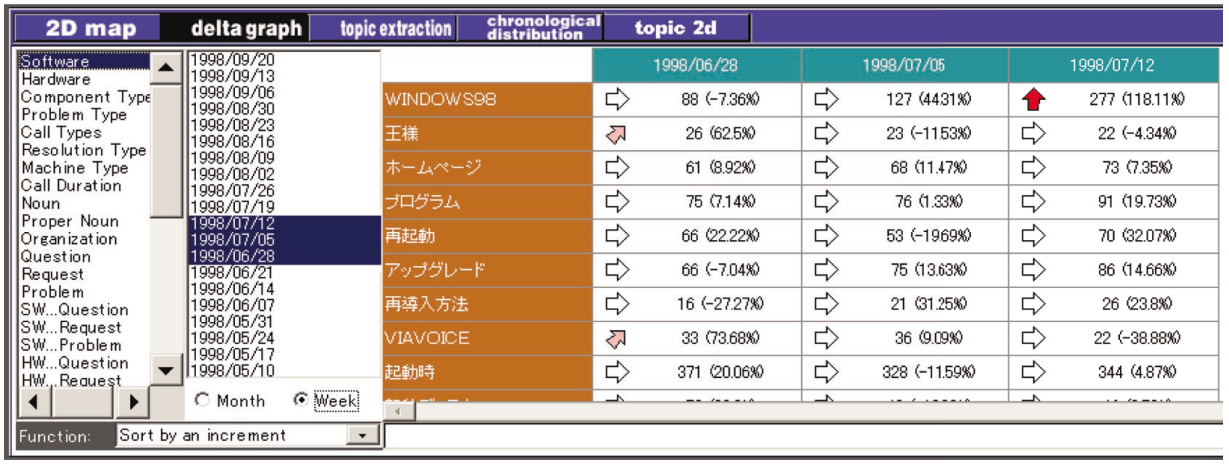
(e) Multiple viewpoints—Different persons want to extract different information from different points of view. The information contained in the customers' contact information is so rich that different kinds of knowledge can be extracted from analyses with different viewpoints. For example:

- The manager of a call center wants to improve productivity, reduce cost, improve customer satisfaction, etc. From this point of view, analysis of call-taker skills and creation of a FAQ (frequently asked question) database is important.
- An analyst for product development wants to improve product quality. Therefore, early detection of product failures is important.
- A planner for marketing may want to develop an attractive product to market, and analysis of customer buying behaviors is essential from this point of view.

(f) Limited vocabulary—The vocabulary is fairly limited (as also described in a later section with Figure 10). For example, out of one month of data (about 40000 text entries) at a PC help center in Japan, we extracted a total of 1473444 noun phrases (including compound nouns and their component words) as keywords. However, since most of the noun phrases appeared in more than one text record, the number of different keywords was only 101987. And the number of keywords that appeared more than once was 36811, which covered 95.6 percent of the whole set of 1473444 keywords. Furthermore, for all of the keywords, 89.1 percent appeared more than nine times in the entire body of text entries. The keywords that appeared at least one thousand times covered 45 percent of the entire group of keywords.

Thus, this application is ideal for our text mining technology. First of all, the limited vocabulary means that a semantic dictionary with a relatively small number of entries, on the order of a few thousand words, may be quite effective in its coverage. This is especially true for statistical analysis that is concerned with frequently repeated events—we can ignore uncommon expressions. For instance, when we installed TAKMI at a PC help center in the United States, we made a list of words extracted from texts in the records sorted by their frequency and asked call analysts in the help center to assign semantic categories such as [software], [hardware], and [problem] to words that they consider important for call analysis as well as assigning their canonical forms to take care of synonym expressions and variations of inconsistency in expressions. As a result, it took two analysts a couple of days to make a semantic dictionary for over six thousand words, which included over 80 percent of the content words in the call records for these data.

Figure 2 Delta analysis for increasing topics in [software] from Japanese PC help center




**Results of application.** Figure 2 shows that Windows 98 was the most rapidly increasing topic in [software] from the middle of June to the beginning of July in 1998 (year, month, day) as a result of using TAKMI on the call records in a Japanese PC help center. From the monthly distribution of the number of reports that mentioned Windows 98 from January 1998 to February 1999, as shown in Figure 3, we can see that the number was rapidly increasing from July to August 1998. Figure 4 shows a list of [software . . . question] pairs in the reports that mentioned Windows 98 in July 1998. It shows the following messages:

Is it possible to install Windows 98?
Does it support Windows 98?
Can I use Windows 98?
Can I upgrade?

This list tells us that most of the customers were asking whether they could install Windows 98 on their machine. In this case, the company prepared an answer to this question and put the information on their World Wide Web home page in order to reduce the number of calls from customers as well as the workload of the call takers for preparing answers.

At the same time, the effects of such actions can be examined by using our system. Figure 5 shows the monthly distribution of the number of reports on customers' calls in which customers asked whether they could install or use Windows 98 on their PCs.

Figure 6 shows a monthly distribution of 1400 calls in the same Japanese PC help center for VoiceType*, the predecessor of ViaVoice* (an IBM voice recognition product). Since this product was old at this time (July 1997–April 1998), the number of calls was decreasing.

The results of the topic extraction function applied to the same data for the category [Call type][20] is shown in Figure 7. This figure indicates that the main focus of the calls shifts from "presale issues (guidance on purchasing)" (July–August) to "general guidance" (October–November), then to "request" (January). The system could not find any noteworthy items in the February–April period. This result seems to reflect the typical life cycle of a product and customers' behavior.

As a result, our system has been enthusiastically accepted by the help center staff since it relieves them of their routine work and improves the quality of their output. Furthermore, it made possible the use of information from large amounts of text data, not only by a limited number of analysts in the help centers, but also by nonanalysts in development and even by executives.

TAKMI has been employed in IBM PC help centers in Japan and the United States, supporting both Japanese and English. Despite the difference in languages, we could apply the same framework to both Japanese and English with a small modification in

Figure 3 Monthly distribution of calls on Windows 98 from Japanese PC help center

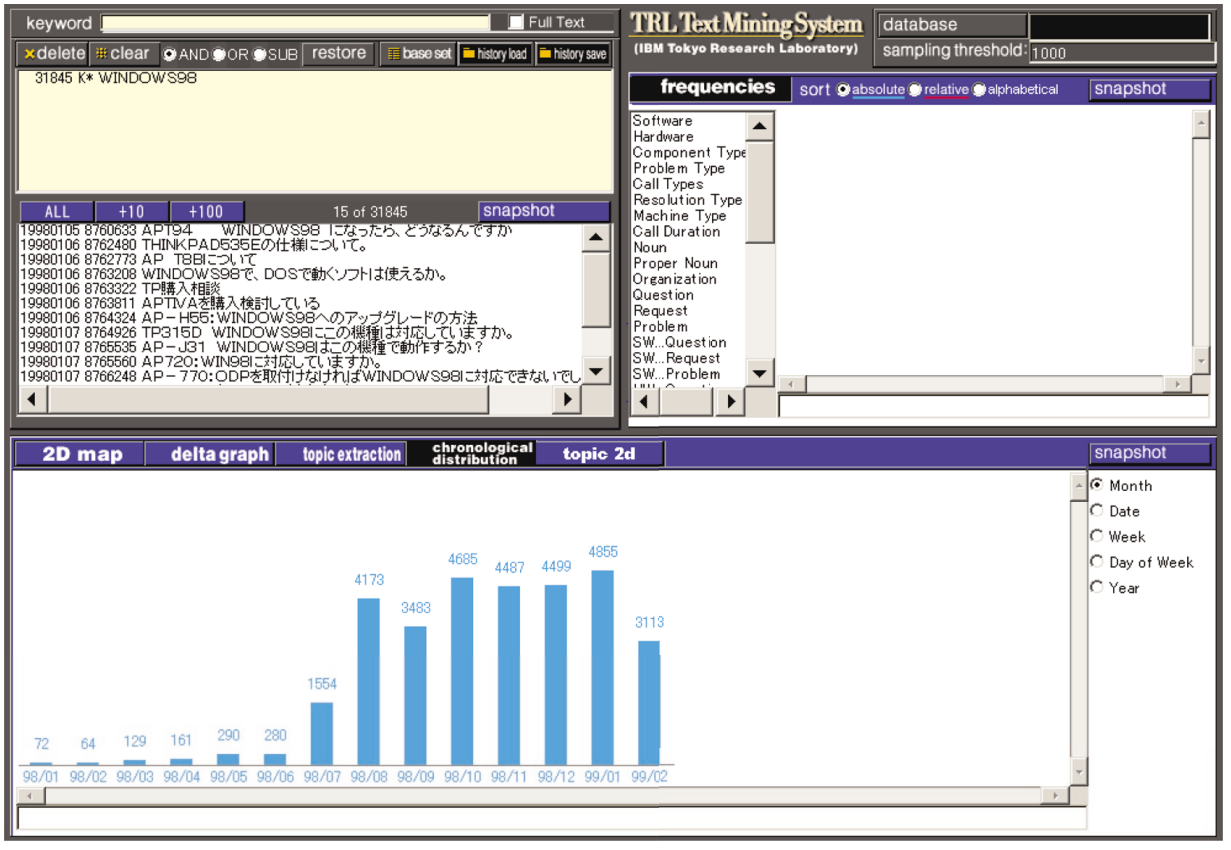
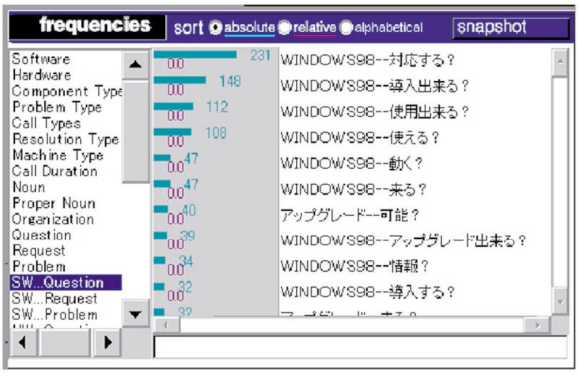



Figure 4 Frequency list of [software...question] pairs in the reports that mentioned Windows 98




the grammatical analysis for recognizing intentions and dependencies. In these help centers, TAKMI has been evaluated as very effective for problem detection as well as for reducing the workload of analysts.

The two-dimensional association analysis proved quite effective for finding problems in products. For example, by checking associations among [machine type] and [problem] in one month of data from the Japanese help center, a specific machine was strongly associated with the concept "slow," and in the data associated with the machine type and "slow," "hard disk" had a high relative frequency value. As a result, the development team found a shielding problem in the hard disk of the specific type of machine that made them operate slowly.

**Evaluation of concept extraction in TAKMI.** We have evaluated the expressive power of the results of concept extraction by using the customer contact records.

Figure 5 Number of calls on the possibility of installing Windows 98

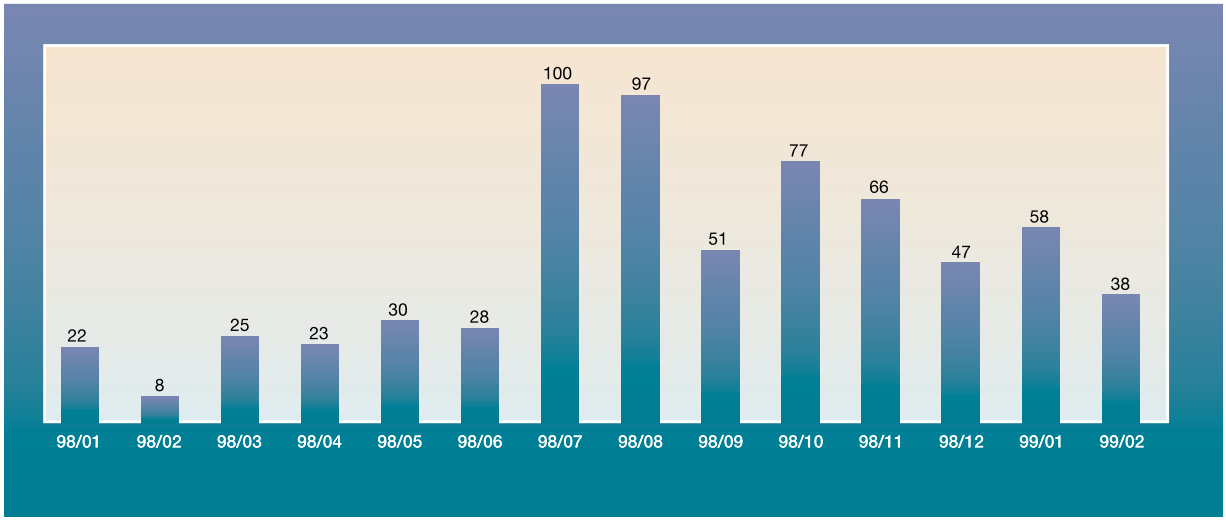
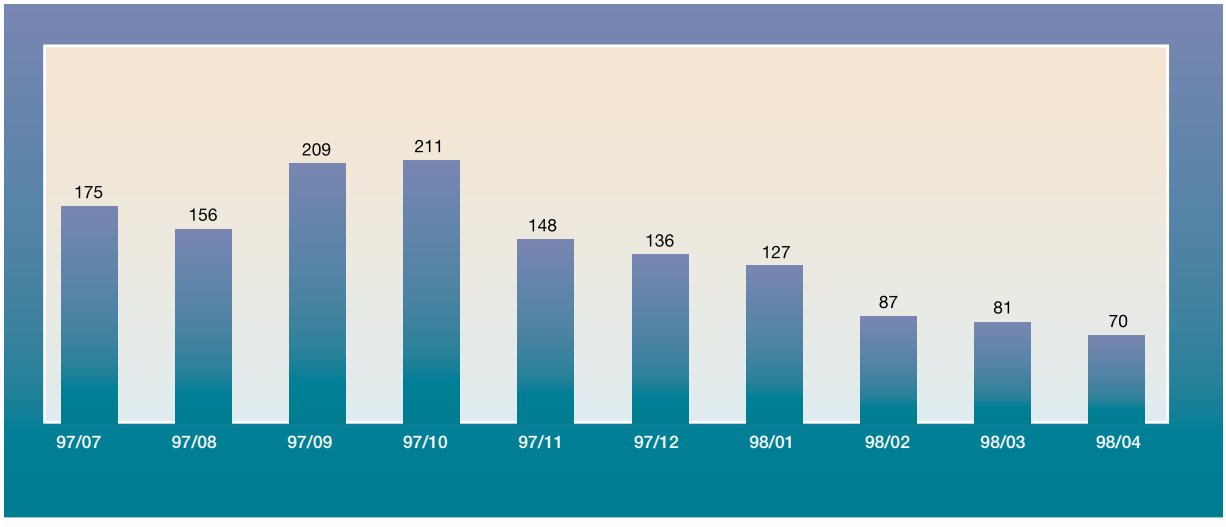



Figure 6 Monthly distribution of calls on VoiceType




First, in order to investigate the practical needs of intention analysis, we analyzed the use of verbs in call records. As shown in Table 3, the verb "use"[21] appeared with expressions that indicate some intentions in over 40 percent of the cases in the call records in the Japanese PC help center. Thus, disregarding this type of information may lead to a relatively high possibility of misinterpreting the textual content.

The complex concept representation based on intention analysis and dependency analysis was quite effective for representing sentential level information described in various expressions. For example, in one month of data from the Japanese PC help center, 55 cases contained "file . . . not found" in the [`software . . . problem`] category. Among the 55 cases, only 13 cases contained the same surface expression to describe that concept. The complex

Figure 7 Result of topic extraction from calls on VoiceType from Japanese PC help center

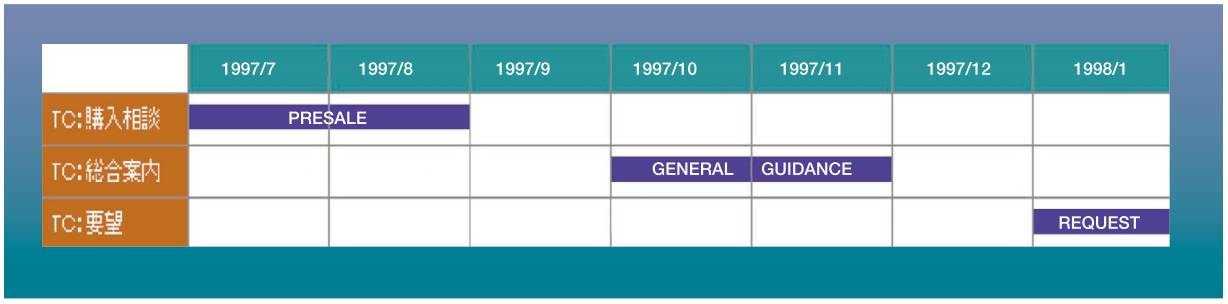



Table 3 Appearances of "use" in call records with their context of intention (originally in Japanese)

| Typical Expression | Indication of Intention | | | | Number of Appearances |
|---|---|---|---|---|---|
| | Possible | Negation | Request | Question | |
| Use | N | N | N | N | 1998 (56.2%) |
| Not possible to use | Y | Y | N | N | 637 (17.9%) |
| Possible to use | Y | N | N | N | 297 (8.4%) |
| Want to use | N | N | Y | N | 262 (7.4%) |
| Is it possible to use . . . ? | Y | N | N | Y | 137 (3.9%) |
| Do/does not use | N | Y | N | N | 137 (3.9%) |
| Does it use . . . ? | N | N | N | Y | 57 (1.6%) |
| Isn't it possible to use . . . ? | Y | Y | N | Y | 19 (0.5%) |
| Others | | | | | 10 (0.3%) |
| Total | | | | | 3554 (100%) |

concept representation has been recognized as especially effective for searching for questions to include in FAQs as a result of its capability for representing sentential level concepts.

In addition, we conducted an experiment to determine whether these complex concepts are better indicators of the categories found in the fixed fields of the call center data. One month of data from the Japanese call center was used. Since categories are assigned by call takers per record, we examined the capability of each form of document representation (noun phrase and verb phrase versus predicate-argument) to classify each document according to the manually assigned categories. We calculated the entropy of each representation for distribution over each item in a category type by using the following formula.

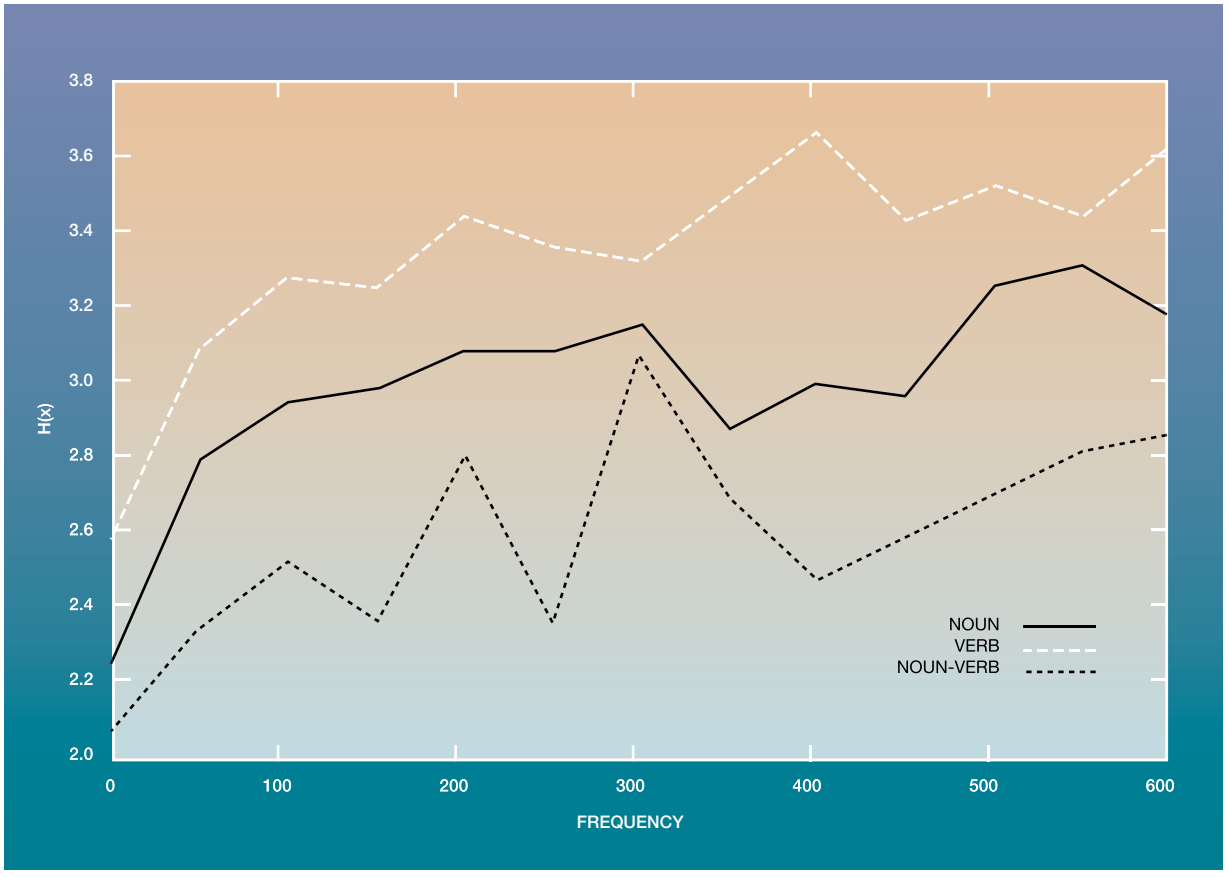
$$H(w_x) = \sum_{i=1}^{N} P(w_x|C_i) \log_2 \frac{1}{P(w_x|C_i)}$$

where $w_x$ is an instance of a representation such as "machine" (noun) or "program . . . delete" (noun-verb). $C_i$ is an item in a category type that has $N$ items such as "Technical QA" and "presale issues" in [Call type]. $P(w_x|C_i)$ is a probability that $w_x$ appears in $C_i$. Thus, lower entropy indicates that the distribution of the representation is a better indicator of specific categories.

Figure 8 shows average entropies for [Call type] over the words with the same frequency within all of the data. Since frequency of items is an important factor in statistical analysis in text mining, we compared the expressive power of items with the same frequency. Figure 9 shows entropies for another category, [Component type], that contain items such as "monitor," "modem," and "hard drive."

In both cases, the entropy of the predicate and argument pair (indicated with "noun-verb") is much lower than just nouns or just verb phrases. Since [Component type] basically indicates objects, the

Figure 8 Average entropies for [Call type]




difference of the entropy between nouns and predicate-argument pairs is smaller than that for [Call type]. However, since it includes some items close to [Call type] such as "installation issues" and "general guidance," predicate-argument pairs are effective for the classification of calls.

To summarize, these results are strong evidence supporting our claim regarding the effectiveness of our representation.

### Application of TAKMI to other data

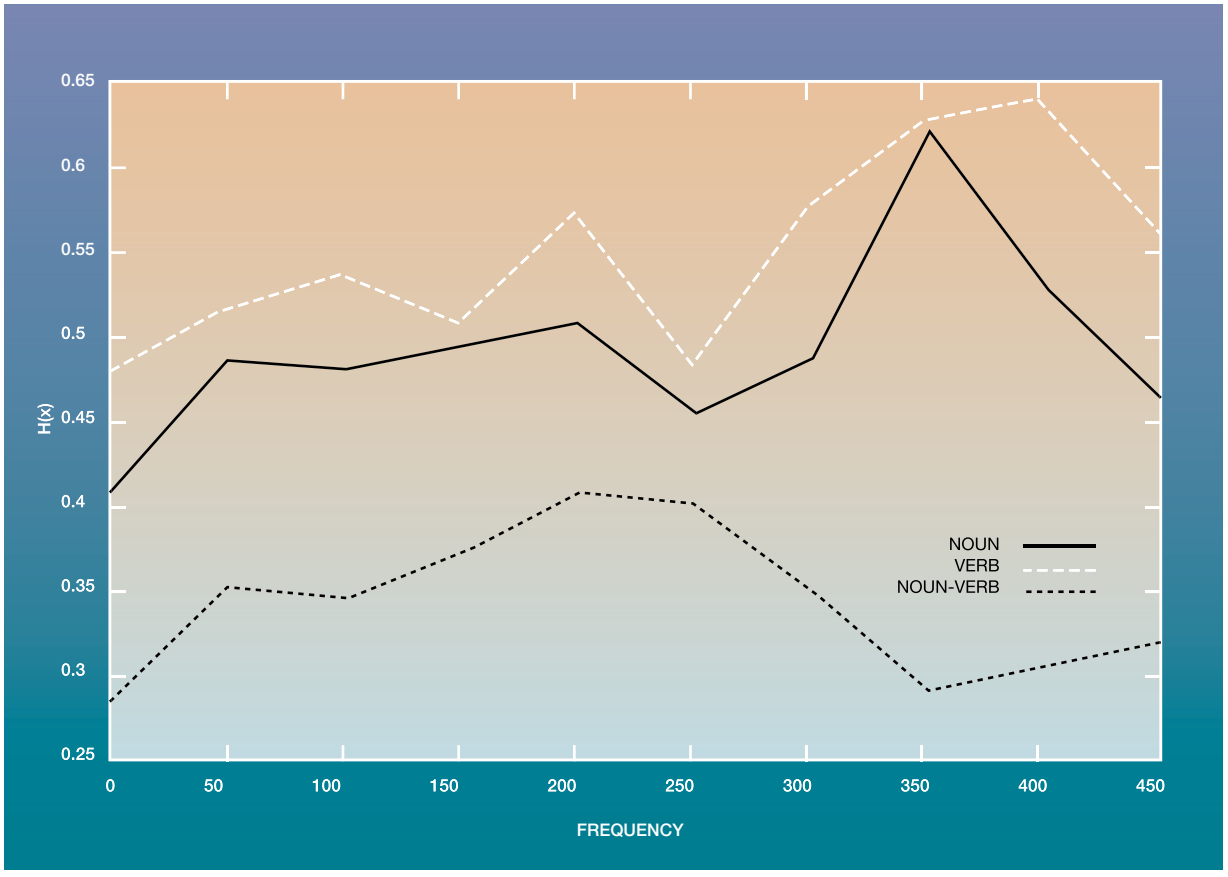
We have verified that our framework is also effective for other data such as patent documents and Medline[22] data.

However, in patent and Medline data, the size of the vocabulary is quite large, unlike the records in contact centers, unless focusing on data in a specific domain. For example, we applied a POS tagger to the following data sets:

- 31440 call records in a PC help center in the U.S. (total 2613215 words)
- 10070 abstracts of patent data (total 1267668 words)
- 15943 abstracts of Medline data (total 2600513 words)

We analyzed the distribution of words that the tagger assigned the POS "unknown" because it could not assign any other POS such as "noun" and "verb" based on grammatical information and its lexicon. Basically, most of the unknown words were technical terms and proper nouns. The unknown POS was assigned to

Figure 9 Average entropies for [Component type]



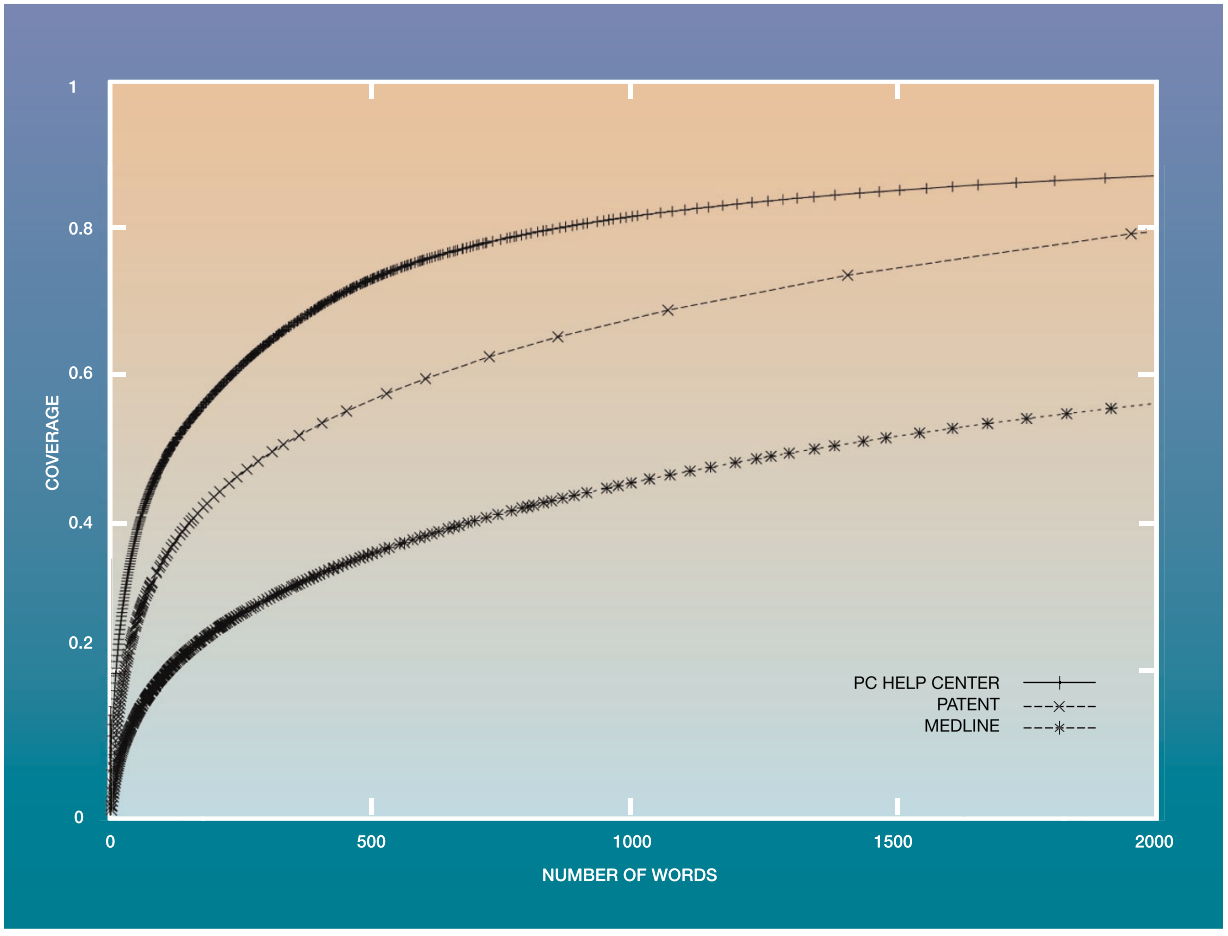

- 17.7 percent of all words in the call records (total 391013 words with 28463 distinct words)
- 2.5 percent of all words in the abstracts of patent data (total 29140 words with 6607 distinct words)
- 11.3 percent of all words in the abstracts of Medline data (total 293963 words with 38047 distinct words)

Even though the total number of unknown words is largest for the call records, those records contain only a small number of frequently appearing unknown words compared to the other two data sets. As shown in Figure 10, the most frequent 1000 words cover over 80 percent of the total word appearances of unknown words in the call records, whereas it takes more than 2000 words to cover 80 percent in the patent abstracts and 8000 words to cover 80 percent for the Medline data. This comparison shows the higher workload for creating semantic dictionaries for patent and Medline data compared to call records.

Yet we can still apply TAKMI for these data sets without creating any new semantic dictionaries. Since the patent and Medline data have tagged information such as titles, the name of the author, and classification codes such as the IPC (International Patent Classification) code, we can use this information as well as using the grammatical category of each word assigned by a POS tagger.

**Analysis of patent documents.** Japanese patent documents are written in a markup language, and basic items such as titles, dates of issue, authors' names, and organization names are explicitly indicated by tags and can easily be extracted by applying simple pattern matching. Thus, from 15000 patent documents, we generated indexes by extracting terms in

Figure 10 Coverage of high frequent unknown words within each data set




the categories shown in Table 4 without human intervention. Among the terms for the categories shown in Table 4, the keywords in G through I were extracted by applying Japanese morphological analysis[23] based on Japanese grammar and lexicons in such a manner that nouns and compound words consisting of nouns were extracted as keywords.

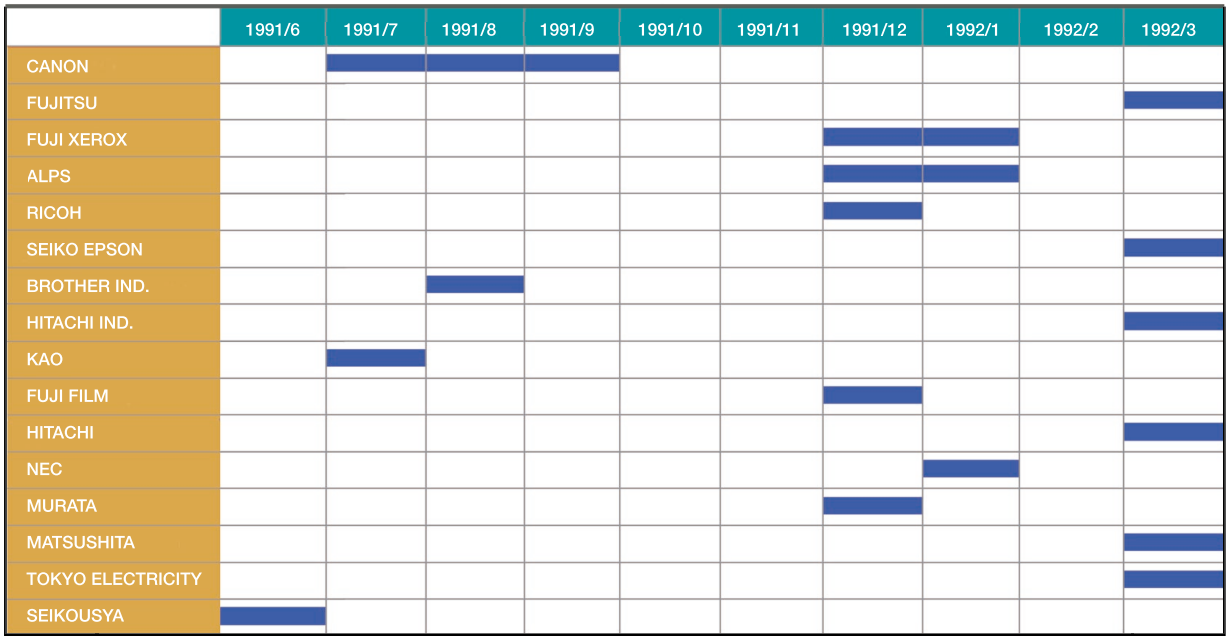
By applying TAKMI to patent documents, we could easily draw a trend map of the patent submissions for a specific technology that shows changes for the entries of each company for the technology as well as specific features of the patent documents of each company.

*Analysis of patent strategy of a specific organization.* For example, by selecting the name of a company that is listed in the categorical viewer as category E in Table 4, a user can focus on the set of patent documents submitted by the company. Then, if the user chooses category G in Table 4 and invokes a topic

Table 4 Example of categories of terms extracted from patent data

| | |
|---|---|
| A | Date of issue |
| B | IPC (International Patent Classification) codes |
| C | Technology fields (derived from IPC) |
| D | Locations |
| E | Organization names |
| F | Patent holders |
| G | Keywords in title fields |
| H | Keywords in fields for the purpose of the inventions |
| I | Keywords in the bodies of a patent document |

Figure 11 Topic extraction in [organization names] from 308 patent documents containing the word "inkjet"




extraction viewer, terms that describe patented technologies are listed in accordance with the period in which the company submitted patents for each technology, showing the frequency of its submissions, since terms are listed according to their topic indexes in a topic extraction viewer, and the topic index is calculated in such a manner that a higher value is assigned to a term that appears frequently in a short period. Thus, this view reveals the patent strategy of the company.

*Analysis of competitive organizations for a specific technology.* By retrieving patent documents with a keyword that indicates a patented technology, a user can focus on a set of patents related to the technology. Then, if the user chooses category E in Table 4 and invokes a topic extraction viewer, organizations, mainly companies, that submitted patents related to the technology are listed in accordance with the period and frequency of their patent submissions. Figure 11 shows that "Canon" frequently submitted patents related to "inkjet" prior to other companies, and that other companies followed Canon's lead.[24] Thus, this figure shows the relative strength of companies with regard to a particular technology. By listing terms with higher relative frequency values as described earlier, we can clarify the features of each set of patented technologies as we select them.

As another example, by retrieving patent documents containing the word "inkjet" and focusing on the same set of 308 patent documents related to this topic, a user can examine technical terms typically related to "inkjet" by choosing category G in Table 4 and invoking a singularity analysis viewer. Then, by choosing the name of a company listed in a categorical viewer with category E, say, "Canon," the user can focus on a set of Canon patents related to "inkjet" technology. The user can also examine technical terms typically related to "inkjet" in patent applications submitted by Canon by invoking a singularity analysis viewer with category G in Table 4. According to our data extracted from the full set of 15 000 patents, the top three terms in Canon patents are "carriage movement device," "movement device," and "jet," whereas the top three terms in Fujitsu patents related to "inkjet" are "inkjet head," "inkjet print head," and "inkjet."

Thus, we can apply TAKMI to any collection of documents even without creating a semantic dictionary. In fact, even in the PC help centers that are using TAKMI with their own semantic dictionaries, grammatical categories such as noun and proper noun (basically unknown words) are used in order to deal with unregistered words such as the name of a virus that suddenly appears. However, creation of a semantic

dictionary is effective in supporting deeper analysis. For example, the data from customer centers in broadcasting companies deal with a very wide range of topics. Nevertheless, we can register words or phrases that analysts are interested in, allowing them to ignore other words to avoid a mixture of data on irrelevant topics. In addition, application of dependency analysis to Medline data allows us to analyze interactions among objects and their effects.

## Concluding remarks

We have developed a framework of text mining to discover knowledge from very large amounts of textual data, especially focusing on NLP to extract concepts from each piece of text. Regarding technical enhancements in NLP, we believe that intention analysis and dependency analysis were the key features of our successful implementation. Intention analysis allowed us to classify predicates by analyzing functional words, which have often been ignored as stop words. Dependency analysis allowed us to capture higher-level sentential information effectively. By applying the results of concept extraction to statistical analysis functions that take advantage of semantic features of concepts, our system, TAKMI, provided practical results. Categorization of terms based on the attached semantic features is important in organizing the output knowledge as well as to facilitate interactive analysis to deal with multiple viewpoints.

Because of the interactive manner in which an analyst can easily confirm the results of analysis by checking the original documents, and the nature of the statistical analysis that tends to ignore minor patterns, occasional failures in NLP can be treated as noise and do not cause critical problems. Thus, even for the informal style of writing in the text entries of PC help center data and even for difficult grammatical analysis, TAKMI generated valuable results.

As a result, our system has been enthusiastically accepted by users who had been manually analyzing textual records, since it eases their routine work and improves the quality of their output.

## Acknowledgments

We would like to thank Jung-kook Hong, Matthew Hurst, Hiroshi Maruyama, Hirofumi Matsuzawa, Masayuki Morohashi, Akiko Murakami, Hiroshi Nomiyama, Koichi Takeda, Hironori Takeuchi, the PC SW Help Center in IBM Japan, the Text Analysis and Language Engineering Group of the IBM Thomas J. Watson Research Center, and the e-Business Technologies Group of the IBM Almaden Research Center for their collaboration and support of the TAKMI project, as well as the anonymous reviewers for their comments and suggestions. We would also like to thank Shannon Jacobs for invaluable help in proofreading an early version of this paper.



## Cited references and notes

1. O. Zamir, O. Etzioni, and R. Karp, "Fast and Intuitive Clustering of Web Documents," *Proceedings of KDD-97* (1997), pp. 287–290.
2. W. Cohen and H. Hirsh, "Joins That Generalize: Text Classification Using WHIRL," *Proceedings of KDD-98* (1998), pp. 169–173.
3. G. Salton and M. J. McGill, *SMART and SIRE Experimental Retrieval Systems*, McGraw-Hill, Inc., New York (1983).
4. A. M. Hearst, "Untangling Text Data Mining," *Proceedings of ACL-99* (1999), pp. 3–10.
5. K. Night, "Mining Online Text," *Communications of the ACM* **42**, No. 11, 58–61 (1999).
6. U. Hahn and K. Schnattinger, "Deep Knowledge Discovery from Natural Language Texts," *Proceedings of KDD-97* (1997), pp. 175–178.
7. *Information Extraction*, Lecture Notes in Artificial Intelligence, M. T. Pazienza, Editor, Springer-Verlag, Rome (1997).
8. Message Understanding Conferences, see http://www.itl.nist.gov/iad/894.02/related_projects/muc/index.html.
9. R. Feldman and I. Dagan, "Knowledge Discovery in Textual Databases," *Proceedings of KDD-95* (1995), pp. 112–117.
10. R. Feldman, W. Kloesgen, and A. Zilberstein, "Visualization Techniques to Explore Data Mining Results for Documents," *Proceedings of KDD-97* (1997), pp. 16–23.
11. B. Lent, R. Agrawal, and R. Srikant, "Discovering Trends in Text Databases," *Proceedings of KDD-97* (1997), pp. 227–230.
12. J. Mladenic, "Text-Learning and Related Intelligent Agents: A Survey," *IEEE Intelligent Systems* **14**, No. 4, 44–54 (1999).
13. V. Hatzivassiloglou and K. McKeown, "Predicting the Semantic Orientation of Adjectives," *Proceedings of ACL-97* (1997), pp. 174–181.
14. H. Matsuzawa and T. Fukuda, "Mining Structured Association Patterns from Databases," *Proceedings of the 4th Pacific and Asia International Conference on Knowledge Discovery and Data Mining* (2000), pp. 233–244.
15. The category of "fail" is very dependent on the domain. Thus, it should be defined in the semantic dictionary.
16. R. Agrawal, T. Imielinski, and A. Swami, "Mining Association Rules Between Sets of Items in Large Databases," *Proceedings of the ACM SIGMOD '93* (1993), pp. 207–216.
17. H. Nomiyama, *Topic Analysis in Newspaper Articles*, Technical Report TR-0129, IBM Tokyo Research Laboratory, Tokyo (1996).
18. M. Morohashi, K. Takeda, H. Nomiyama, and H. Maruyama, "Information Outlining—Filling the Gap Between Visualization and Navigation in Digital Libraries," *Proceedings of the International Symposium on Digital Libraries* (1995), pp. 151–158.

19. P. Xia, "Knowledge Discovery in Integrated Call Centers: A Framework for Effective Customer-Driven Marketing," *Proceedings of KDD-97* (1997), pp. 279–282.
20. This category is contained in structured data, whereas calls on VoiceType were collected based on information in unstructured text.
21. The verb "use" is "tsukau" in Japanese.
22. Information on Medline can be found at http://www.nlm.nih.gov/.
23. H. Maruyama, *A Formal Approach to Japanese Analysis in Japanese-to-English Machine Translation*, Dissertation, Kyoto University, Kyoto, Japan (1995).
24. This is a result of analysis in a small set of sample data to demonstrate the capability of the system.

*Accepted for publication June 26, 2001.*

**Tetsuya Nasukawa** *IBM Research Division, Tokyo Research Laboratory, 1623-14, Shimotsuruma, Yamato, Kanagawa, Japan (electronic mail: nasukawa@jp.ibm.com).* Dr. Nasukawa joined the IBM Tokyo Research Laboratory in 1989, after receiving a master's degree from Waseda University. He was involved with English-to-Japanese machine translation projects and digital library projects before he started the text mining project in 1998. He received a Ph.D. degree from Waseda University in 1998 for his work on natural language processing. His research interests include natural language understanding.

**Tohru Nagano** *IBM Research Division, Tokyo Research Laboratory, 1623-14, Shimotsuruma, Yamato, Kanagawa, Japan (electronic mail: tohru3@jp.ibm.com).* Mr. Nagano joined the IBM Tokyo Research Laboratory and text mining project in 1998, after receiving a master's degree in computer science at the University of Tsukuba in Japan. Currently he is a member of the text mining project of the Internet Technology Division, where he is developing applications for the text mining system. His research interests include natural language processing, machine learning, and statistical analysis.

# ABLE: A toolkit for building multiagent autonomic systems

by J. P. Bigus
D. A. Schlosnagle
J. R. Pilgrim
W. N. Mills III
Y. Diao

This paper describes a toolkit for building multiagent autonomic systems. The IBM Agent Building and Learning Environment (ABLE) provides a lightweight Java™ agent framework, a comprehensive JavaBeans™ library of intelligent software components, a set of development and test tools, and an agent platform. We describe a series of agents built using ABLE components and present three case studies of applications using the ABLE toolkit. The Autotune agent is a closed-loop controller agent that supports hierarchical distributed control. The Subsumption agent defines specific behaviors or strategies and can be plugged into a multiagent subsumption infrastructure. The Autonomic agent architecture features sensors and effectors for interacting with the external environment, layers of reflexive, reactive, and adaptive subsumption agents, components that dynamically model the autonomic system itself and its environment, and components for emotions, planning, and executive-level decision-making. By using the ABLE component library to build agents running on the ABLE distributed agent platform, we discuss how we can incrementally add new behaviors and capabilities to intelligent, autonomic systems.

It has been over 50 years since Alan Turing described the prototypical test for intelligent machines. In Turing's view, a computer could be called intelligent if it could pass as a human while conversing via a computer terminal. For researchers delving into the mysteries of human and machine intelligence, the so-called Turing Test has been both an inspiration and a millstone around their necks since that time.

The field of artificial intelligence (AI) started out with great hopes and fanfare, beginning with the Dartmouth conference on AI in 1956. The next decade saw an exploration of both symbolic and neural network approaches to knowledge representation, reasoning, and machine learning. By 1970, all was going so well that Marvin Minsky declared in a *Life Magazine* article, "In from three to eight years we will have a machine with the general intelligence of an average human being. I mean a machine that will be able to read Shakespeare, grease a car, play office politics, tell a joke, have a fight. At that point the machine will begin to educate itself with fantastic speed. In a few months it will be at genius level and a few months after that its powers will be incalculable." Although his vision may still be valid, his timing was off by several decades. Even though substantial progress has been made on the pieces of intelligence—speech recognition, natural language understanding, knowledge representation, machine reasoning, machine learning, emotion, and speech generation—putting the pieces together to create human-level intelligence has proven to be difficult.




0018-8670/02/$5.00 © 2002 IBM

The artificial intelligence technology pendulum has swung from whole-hearted devotion to symbol-processing techniques, to reactionary forays into neural networks and other subsymbolic approaches, and on to biologically inspired genetic algorithms and fuzzy reasoning. Today, most researchers admit that a combination of technical approaches must be used to achieve human-level performance. In *The Society of Mind*, Minsky described a set of mechanisms that he called mental agents that operate in parallel and that compete and cooperate to yield human intelligence.[1] In his subsumption architecture, Brooks describes an architecture of behavioral layers that provides robust function and supports reactive behaviors in mechanical robots.[2] More recently, papers by Sloman and by Caulfield and Johnson have described architectures for consciousness or self-aware systems that rely on layered architectures with emotional components.[3,4]

Clearly, a monolithic software architecture using a single technology will not bring us closer to our goal. In our work, we are exploring an incremental approach for developing intelligent autonomic systems—systems that have self-awareness and can reason about their internal components and state.[5] Autonomic systems must adapt to environmental changes and strive to improve their performance over time. They must be robust and be able to routinely overcome internal component failures. Autonomic systems must interact and communicate with other systems in a heterogeneous computing infrastructure. Our approach to building autonomic systems is based on combining autonomous intelligent agents in a well-structured way. This approach mirrors the structure of the human brain wherein there are clearly defined, function-specific processing centers connected by forward and backward communication channels and adaptive feedback loops.

In this paper, we briefly describe an architecture that combines elements of these approaches and melds them into a coherent, scalable architecture that we believe will lead to robust deployed autonomic systems. These systems rely on sensors to obtain input from the world and effectors to take action and make changes to the world. There are memory components providing short-term, long-term, and associative memory functions. There are reflexive, reactive, and goal-oriented proactive components. There are components for reasoning, planning, and learning new behaviors from interactions with the world. There is an emotional component that associates feelings with internal states and influences decision-making and learning processes. This architecture reflects much of what is known about how people think and process information, including the role emotions play in our reasoning.[6,7]

This paper is organized as follows. First, we describe the Agent Building and Learning Environment (ABLE), a software architecture and framework, component library, development tooling, and agent platform for constructing autonomous intelligent agents and multiagent systems. We then present two application case studies, a system administration application using multiple agents, and a diagnostic application. Next, we describe several derivative ABLE agents including the Autotune control agent, a Subsumption agent, and an Autonomic agent that is an ABLE-based architecture for incrementally building autonomic systems. We then discuss the current status of ABLE and our plans for enhancing the toolkit and for implementing our autonomic system architecture.

## Agent building and learning environment

The recent surge of interest in software agents has prompted a corresponding increase in toolkits for constructing them. Although many projects use a "roll your own" approach in which each agent is uniquely hand-coded, there are benefits to using a component-based approach. The Java** language has several characteristics that make it an ideal platform for implementing agents: code portability resulting from its use of a standard virtual machine, support for object-oriented programming techniques, native support for multithreading, and introspection of object properties and methods. In addition, the JavaBeans** component specification enables the creation of reusable Java components with well-defined interfaces and behaviors.

Quite a few agent toolkits and multiagent platforms are available for both educational and commercial use. The CIAgent framework developed by one of the authors is a lightweight agent framework written in the Java language and intended for educational use.[8] The Java Agent Template Lite (JatLite), developed at Stanford University, is focused on communications-related issues of agent systems. The IBM AGLETS* mobile agent framework, now an open source project, provides a Java platform for creating mobile agent applications. AgentBuilder** from Reticular Systems (a part of IntelliOne Technologies), is an integrated software development toolkit for constructing belief-desire-intention (BDI) agents

in Java. The ZEUS agent-building toolkit developed by British Telecommunications plc features a Java component library with a planning and scheduling system, support for multiple interaction protocols, and a set of tools for building agents. The Foundation for Intelligent Physical Agents (FIPA), an international standards body working for interoperability between agents and agent platforms, has defined specifications for agents, agent management services, and agent communications languages.[9] Several projects implement FIPA compliant agent platforms. The FIPA Open Source (FIPA-OS), developed by Nortel Networks, is an open source implementation of the FIPA agent communication language and agent platform. The Java Agent DEvelopment (JADE) framework, developed at CSELT S.p.A. (now Telecom Italia Lab, or TILab) is another FIPA-compliant multiagent toolkit. For a more detailed overview of these agent environments, see Bigus and Bigus.[8]

The Agent Building and Learning Environment[10] is a Java-based toolkit for developing and deploying hybrid intelligent agent applications. Hybrid approaches synergistically draw on the strengths of each technology while compensating for any weaknesses. For example, rules have the advantage of explicitly defined knowledge, but they can be brittle and inflexible. Neural networks, in contrast, can adapt or learn from inputs, but the learned knowledge is often difficult to make explicit. The value of combining multiple techniques such as neural network learning with rule-based inferencing has been demonstrated by prior work.[11]

The ABLE toolkit was designed to provide a fast, reusable, and scalable architecture for the construction of intelligent software components and agents. A fundamental design philosophy of ABLE is that successful intelligent agents will require multiple reasoning and learning techniques. ABLE builds on the standard JavaBeans model by defining a lightweight framework for agent behavior. ABLE has a component library of data access, machine learning, machine reasoning, and optimization algorithms packaged as JavaBeans, known as AbleBeans. ABLE provides a Java Swing-based GUI (graphical user interface) for creating and configuring AbleBeans, and for constructing and testing the agents built from them. ABLE also provides an agent platform for deploying agents across a distributed computing system. By building a comprehensive suite of intelligent JavaBeans and tooling for easily combining and connecting those beans, ABLE permits developers to explore the applications of software agents and their behaviors in distributed multiagent systems. The ABLE toolkit has been available for downloading from the IBM alphaWorks* site since May 2000.[12] In the following sections, we describe the fundamental design and architectural attributes of ABLE.

## ABLE agent framework

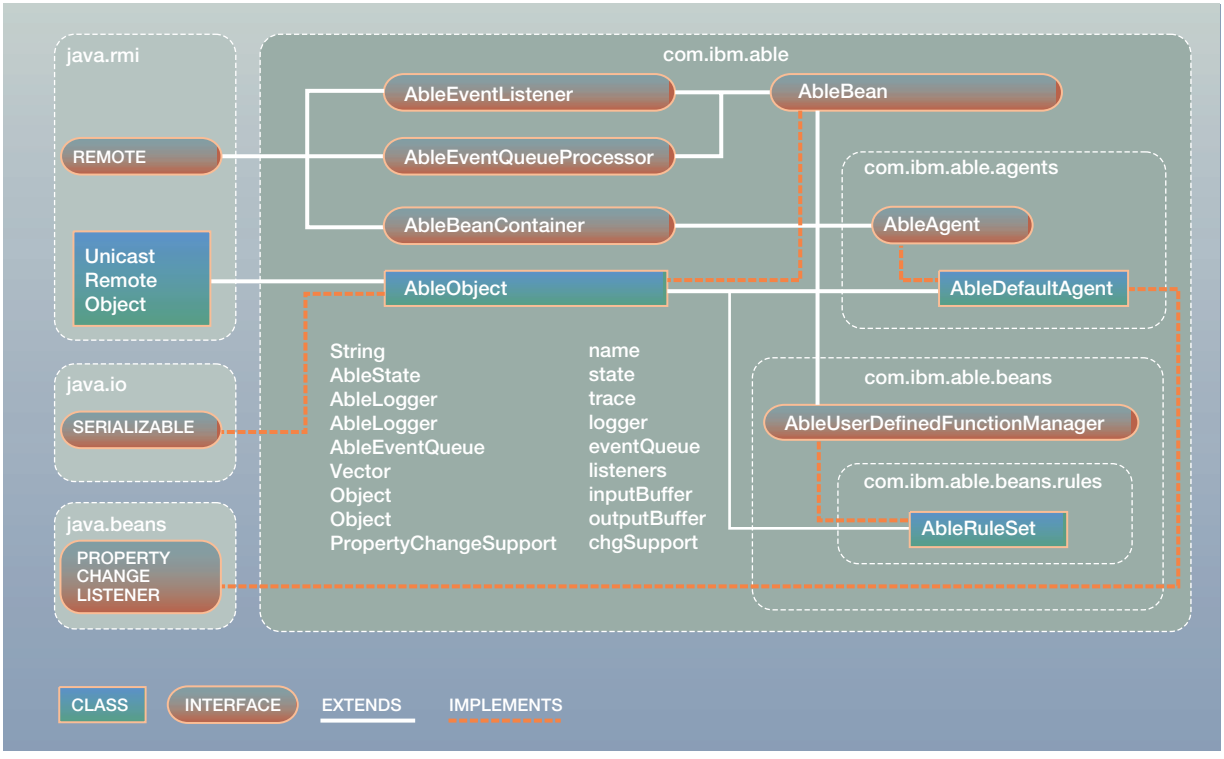
The ABLE agent framework is a lightweight software architecture that allows algorithms to be packaged as JavaBeans that can be deployed as standard Java components or as autonomous agents. Figure 1 shows the set of Java interfaces and base classes comprising the ABLE framework.

AbleBeans are standard JavaBeans components used in the ABLE framework. The AbleBean Java interface defines a set of common attributes (name, comment, state, etc.) and behavior (standard processing methods such as init( ), reset( ), process( ), quit( )), allowing AbleBeans to be connected to form AbleAgents. AbleBeans are connected using three fundamentally different methods: data flow, events, and properties.

Data-flow or buffer connections are used to wire together AbleBeans using a data-flow metaphor. Each AbleBean can have an input buffer and an output buffer that are implemented as Java Objects. A set of AbleBeans can be connected by buffer connections forming a directed, acyclic graph. The set of AbleBeans is then processed in sequence starting at the root of the tree. Each AbleBean takes the data from its input buffer, processes the data, and places the data in its output buffer. This data-flow mechanism is extremely fast and is very useful for applications such as neural networks that have a natural data-flow processing paradigm.

Event connections are used to register an object as a listener on an AbleBean. AbleBeans support synchronous and asynchronous event processing using AbleEvents, which extend the Java EventObject class. Each AbleBean has an event queue on which it receives AbleEvent notifications or action requests to be processed. Each AbleEvent contains a Boolean flag that indicates whether the event should be handled synchronously on the caller's thread or asynchronously by placing it on the receiver's event queue and processing it on a separate thread. Every AbleEvent can be used as either a data event with an associated eventId (event identifier) and data object, or as an action event, with an associated action string and data object. Data notification events hold

Figure 1 ABLE agent framework classes and interfaces




data and allow AbleBeans to inform one another of complex state changes. Action events allow method-invocation with arguments and contain an action field that maps to a method name on the notified AbleBean. The AbleEvent argument object is also passed to the method on the receiving AbleBean. Thus, any method can be called on a listening AbleBean through this mechanism. Although event processing adds some overhead, it is more flexible than hard-coded method calls between AbleBeans.

Property connections are used to synchronize two different properties residing in two different AbleBeans. Whenever the first property value is changed via a setter method, the second property on the second bean is also changed via its setter method.

AbleBeans use Java serialization for persistence. All data-flow, event, and property connections are preserved during the passivation and activation cycles. The ABLE run-time environment has a well-defined set of properties that enable serialized AbleAgents to be portable.

**AbleEvents.** Figure 2 shows the data fields in the AbleEvent class. The AbleEvent class provides the means to send data between agents, to request actions to be performed by other agents, or to request transactions with results returned either to the original requesting agent or to some other agent. This design allows the ABLE event-processing infrastructure to be used to implement a variety of agent interaction models. For example, a dialog between two agents can be supported by exchanges of AbleEvents where the action holds the request and the replyTo and replyWith fields are used to correlate the responses. Alternatively, an intermediary agent could send a request to one agent and have that agent send its response to a third agent or back to the original requester. Another scenario is to have an agent broadcast its response to an event to multiple agents by specifying a list of agents on the replyTo field. Or, a single agent can send out requests to multiple agents, and use the transactionID field to correlate the responses to the original requests. To summarize, the base ABLE event-processing framework can

Figure 2 The AbleEvent class

```
AbleEvent(
   Object source,          // the source object or sender of the event
   int id,                 // type=ACTION, DATACHANGED, EOF, TRANSACTION
   String action,          // name of the action method
   Object arg,             // passed to action method
   boolean async,          // put on queue (true) or run on caller's thread (false)
   Object replyTo,         // null, single bean, list of beans
   String replyAction,     // used as action in reply event
   String transactionId    // used as work identifier (copied to reply)
)
```

be used to implement almost any agent communication design pattern.

**AbleObject.** The AbleObject class provides a base implementation of the AbleBean interface, defining the standard behavior for all AbleBeans provided with the ABLE toolkit. The AbleObject class extends Java UnicastRemoteObject and implements the AbleEventListener, AbleBean, and AbleEventQueueProcessor remote interfaces. It contains an instance of an AbleEventQueue that handles the optional autonomous timer facility as well as asynchronous event-processing functions for the bean. The timer allows an AbleBean to run autonomously by periodically going to sleep and then waking up to see whether anything needs processing.

The AbleEventListener interface defines two main processing methods: processAbleEvent( ) and handleAbleEvent( ). The first method takes an AbleEvent as an argument, examines its synchronous/asynchronous Boolean flag and processes it accordingly. The second method unconditionally processes the event in a synchronous manner. Default behavior is provided to interpret the action string as a method name and to invoke a method with that name on the bean, passing the argument object as a parameter.

Figure 3 shows the source code for an example AbleBean that extends the AbleObject base class. We import the *com.ibm.able* package and provide a no-argument constructor. The major methods that must be overridden include init( ), which performs one-time initialization; process( ), which is the method called to process the input buffers; and processTimerEvent( ), which is the method invoked when the bean is configured to run as an autonomous agent.

**AbleAgent.** One of the major decisions when creating an agent construction environment is the granularity of the agents. Many projects consider belief-desire-intention (BDI) agents to be the base line. In ABLE, we chose a model where the basic building blocks are functional software components but not complete agents. By making this choice, we can create customized agents with functionality and complexity suitable for their intended use.

The AbleDefaultAgent class provides a default implementation of the AbleAgent interface and defines the standard behavior for all AbleAgents provided with the ABLE framework. The AbleDefaultAgent class extends Java UnicastRemoteObject and AbleObject and implements both the AbleAgent and AbleBeanContainer remote interfaces.

AbleAgents are AbleBeans that are also containers for other AbleBeans. An AbleAgent has its own thread for processing events asynchronously. AbleAgents provide a useful abstraction for packaging a set of AbleBeans wired together to perform a specific function. This function is then available to other AbleBeans or AbleAgents through synchronous process( ) calls or through asynchronous event processing.

AbleAgents extend the AbleObject base class and implement the AbleBeanContainer and AbleUserDefinedFunctionManager interfaces. The AbleBeanContainer allows an AbleAgent to contain other AbleBeans and even other AbleAgents. This pow-

Figure 3 Sample AbleBean Java source code

```
import com.ibm.able.*;
public class SampleAbleBean extends AbleObject implements Serializable {

  public SampleAbleBean() throws RemoteException {
    // set processing options, data flow, timer, etc.
    this("SampleBean");  }

  public void init() throws RemoteException {
    // need to initialize state of this bean, algorithm vars, etc. -- do ONE TIME initializations
    // initialize asynchronous Timer (if used) and define Event processing behavior
  }

  public void process() throws RemoteException {
    // perform synchronous processing on caller's thread
  }

  public void processTimerEvent() throws RemoteException {
    // perform autonomous (asynchronous) processing on own thread
  }
}
```

erful design pattern allows extremely complex agents to be built from subagents and is exploited in our component library implementation. The AbleUserDefinedFunctionManager allows external software to be integrated with AbleAgents as sensors and effectors.

Note that alternate AbleAgent implementations could be developed with behaviors different from the AbleDefaultAgent. For example, we provide an AbleDefaultFIPAAgent that is an AbleAgent that provides all of the required FIPA agent behaviors, and the AutotuneAgent that enables hierarchical distributed control. We discuss the AutotuneAgent and additional AbleAgents in more detail later in this paper.

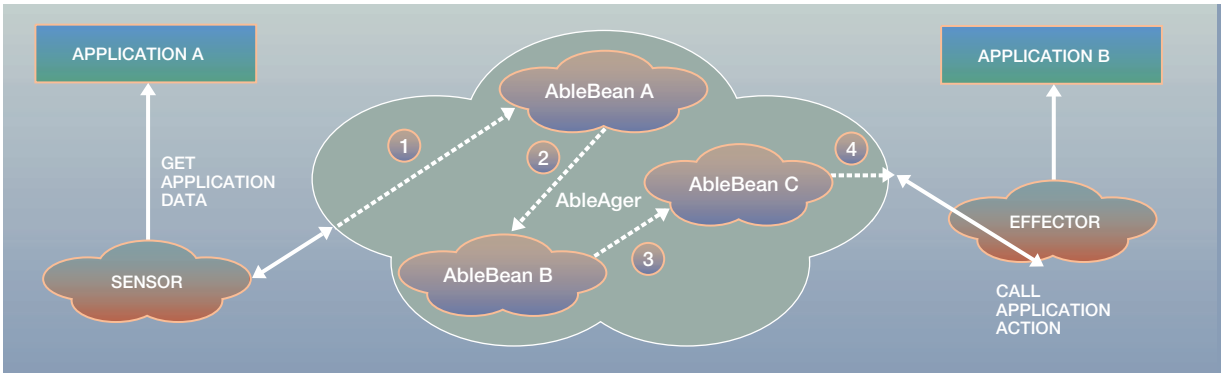
AbleAgents are situated in their environment through the use of sensors and effectors. In ABLE, sensors and effectors are AbleUserDefinedFunction objects that map to method calls on external Java objects. These methods usually call other application programming interfaces (APIs) to either obtain data (sensors) or take actions (effectors). AbleAgents are managers for sensors and effectors, and any contained AbleBeans can invoke those sensors and effectors. Sensors and effectors take arbitrary argument lists and return Java Objects to the caller.

A common scenario is for an AbleAgent to contain one or more beans that reference sensors and effectors. For example, in Figure 4, the AbleAgent contains three AbleBeans, a single sensor, and a single effector. AbleBean A first calls the sensor and obtains data from Application A. It processes the data and passes information to AbleBean B either through a direct method call, an event, or a property connection. AbleBean B processes these data and passes the data on to AbleBean C, which in turn invokes the effector, resulting in a method call on Application B.

## ABLE component library

A fundamental piece of the ABLE system is the component library of AbleBeans. These include data access and filtering beans, machine learning algorithms, machine reasoning and inference engines, and higher-level data mining agents comprised of one or more core beans. In addition, ABLE contains a set of data type classes, defining Boolean, Categorical, Discrete, Numeric, and String literals, variables, and fields. This common data model is used by the beans in the component library. The set of core AbleBeans provided with the ABLE framework includes data beans, learning beans, and rule beans.

Figure 4 Example ABLE agent




**Data beans.** Data access and transformation beans are used to manipulate data for training and testing the learning and reasoning beans. They include:

- *Import*—reads space-, comma-, or tab-delimited data from flat text files
- *DBImport*—reads data from relational databases using JDBC** (JavaBeans Database Connectivity)
- *DataTable*—provides a view over an Import data set, with selected rows and columns
- *Filter*—filters, transforms, and scales data using translate template specifications
- *TimeSeriesFilter*—caches sequential data for use in time-series prediction
- *Export*—writes space-, comma-, or tab-delimited data to flat text files
- *DBExport*—writes data to relational databases using JDBC

**Learning beans.** The learning beans implement several different learning algorithms that can be combined with the data beans to provide lightweight data mining capabilities. They are:

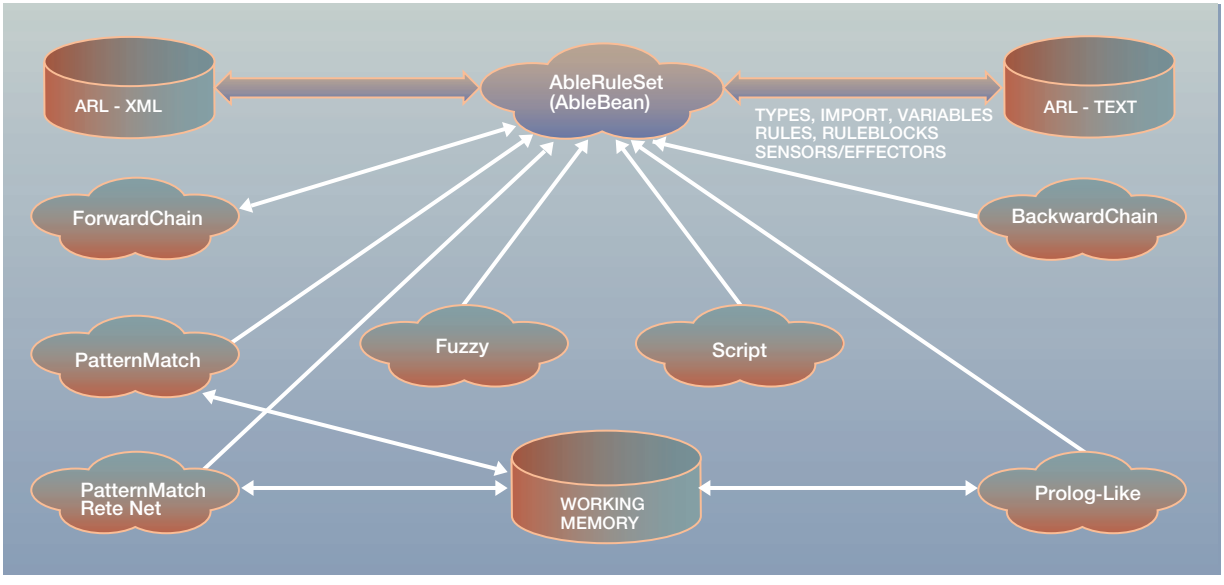
- *Back Propagation*—implements an enhanced back propagation algorithm with pattern and batch updates, hidden layer and output layer recurrence
- *Self-Organizing Map*—supports pattern and batch updates, and a Gaussian neighborhood function
- *Temporal Difference Learning*—supports sequence learning using a reinforcement learning algorithm
- *Radial Basis Function*—supports regression and classification using multiple basis functions with automatic Self-Organizing Map clustering of hidden layer weights
- *Naive Bayes Classifier*—supports incremental learning of discretized data using a Bayes statistics approach
- *Decision Tree*—supports tree-based classification of discretized data using the C4.5 algorithm

**Rule beans.** The ABLE Rule Language (ARL) defines a rich set of rule-based knowledge representation formats including scripting using simple assignments, if-then and if-then-else rules, when-do pattern match rules, and predicate style rules. ARL supports ruleblocks that are named groups of rules similar to macros. ABLE provides a wide range of inference engines to process the ARL rulesets.

As illustrated in Figure 5, ABLE Rule Language can be represented in text or Extensible Markup Language (XML) formats. The AbleRuleSet class parses the text or XML source into a set of AbleRuleBlock and AbleRule objects and instantiates an associated inference engine based on the inference method specified in the RuleSet. The ARL supports very tight integration with Java classes and objects allowing instantiation, access to data members on objects, and invocation of methods on objects from rules. Processors include:

- *Boolean forward chaining*—processes if-then rules using forward chaining
- *Boolean backward chaining*—processes if-then rules using backward chaining
- *Fuzzy forward chaining*—processes if-then rules containing linguistic variables and hedges and several types of fuzzy sets, and supports multistep chaining

Figure 5 AbleRuleSet bean and inference engines




- *Pattern Match engine*—processes when-do pattern match rules using forward chaining against a working memory
- *Pattern Match network*—processes when-do pattern match rules using the Reté network forward chaining algorithm against a working memory
- *Predicate engine*—processes predicate rules using a backchaining algorithm with backtracking (similar to Prolog)
- *Scripting engine*—processes assignments, if-then, if-then-else, while-do, and do-while rules in sequential order

Having a single ABLE Rule Language with pluggable inference engines provides many advantages. A single rule authoring and debugging environment can be used for multiple styles of inferencing. The same knowledge representation can be used for scripting agent behavior, dynamically constructing and configuring agents, and explicitly representing domain knowledge. Alternate implementations of the inference engines can be developed for use in specialized environments where memory or processing resources are constrained.

One of the most powerful aspects of the ABLE Rule Language design is the ability to seamlessly mix symbolic rule-based reasoning with subsymbolic neural network and other machine learning algorithms. A common view is that the subconscious processes of the human brain correlate to neural network approaches and that conscious thought is similar to symbol processing. A single ABLE rule set can reason symbolically about the outputs of multiple neural components. For example, sensory data can be fed into a neural network for clustering into similar groups, for classification into categories, or for prediction of trends. Rules can then process the outputs of the neural network, assign semantic labels to those outputs, and reason about the outputs. Rules could decide to kick off a learning episode in one or more neural networks or to take overt actions to change the external environment.

The ability of rules to invoke other AbleRuleSet beans allows hierarchical configuration and natural partitioning of knowledge into individual rulesets. This ability eases the burden on rule authoring and maintenance. The example AbleRuleSet in Figure 6 shows the Java-like syntax and structure of the ABLE Rule Language. Arbitrary Java classes can be imported into a ruleset, domain-specific function libraries can be loaded, and a variety of built-in and imported variable types can be defined and instantiated in the variables section. Data are passed into and out of the ruleset bean via the input and output statements. Rule methods or ruleblocks can be used to

Figure 6 A sample AbleRuleSet

```
ruleset AbleScriptExample {
   import com.ibm.myClass;   // use myClass as data type in ruleset
   library com.ibm.myLibrary; // each public method becomes a function

   variables {
      myClass myTypeVar = new myClass();  // creates an instance
      myClass myTypeVar2 = new myClass("name", "age", "whatever");
      Object BeanVar = new Object();
      Object Result = new Object();
   }
   inputs { myTypeVar } ;
   outputs { Result };

   void init() {
      // one time initialization rules here
   }

   void main() using Script {
      A1:  ObjectVar = createInstance("com.ibm.able.beans.rules.AbleBooleanRuleSet");
      A2:  ObjectVar.name =  "myRuleSet";
      A3:  Result = instantiateFrom(ObjectVar, "d:\\joe\\myRuleSet.arl");
      A4:  BeanVar = getBean(parent, "aBeanName");
      A5:  Result = init( parent, "aBeanName");
      A6:  Result = processBean( BeanVar);
   }
}

void idle() {
   // idle rules run when the main ruleblock quiesces
}
```

define one-time initialization rules (the init( ) block). The main( ) block allows the user to specify the inference engine that will process the rules. The order of evaluation is entirely determined by the inference engine that is selected. When the main( ) block completes, the optional idle( ) block is run.

**Function-specific AbleAgents.** In addition to the core AbleBeans, the ABLE component library provides a set of function-specific AbleAgents. Data and learning AbleBeans are combined to create neural classification, neural clustering, and neural prediction agents that can be used for lightweight data mining tasks. The set of standard function-specific agents provided with the ABLE framework includes:

- *Genetic search agent*—manipulates a population of genetic objects that may include AbleBeans
- *Neural classifier agent*—uses back propagation to classify data
- *Neural clustering agent*—uses self-organizing maps to cluster or segment data
- *Neural prediction agent*—uses back propagation to build regression models
- *Script agent*—uses the ABLE rule language to define complete agent behavior
- *JavaScript** agent*—uses JavaScript to define agent behavior

The NeuralPredictionAgent uses two Import beans to read training and test data from text files, two Filters to preprocess and postprocess the data, and a back propagation neural network to perform the regression function (shown later in Figure 8). The agent provides high-level functionality through its Customizer as it orchestrates the operation of the five Able-

Beans it contains. The user only needs to specify the source data files and corresponding meta-data files. When initialized, the NeuralPredictionAgent scans the source data and automatically generates the scaling and transformation templates used by the Filters to preprocess and postprocess the data passed through the neural network. Based on the number of input and output fields and their data representation, the neural network architecture is automatically configured. Data-flow connections between the AbleBeans pass data from the Import through the input Filter, through the neural network, and then through the output Filter. The user also specifies target error rates and the maximum number of training epochs. The asynchronous thread of the AbleAgent is used to automatically train the prediction model on a separate background thread and halts when the user-defined termination conditions are met.

Once trained, the NeuralPredictionAgent can be used to process data synchronously. If the model becomes stale after it is deployed, the application could easily force a retraining of the prediction agent, because the training process is automated.

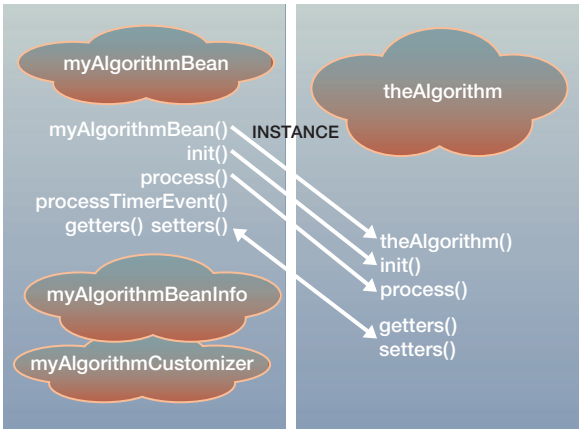
The NeuralClassifierAgent and NeuralClusteringAgent are constructed using multiple AbleBeans in a manner similar to the prediction agent but provide classification and clustering functions, respectively. The neural learning agents provide basic data mining functionality for use in other AbleBeans, giving agents the ability to segment, classify, and make predictions about their environments.[13]

## Extending ABLE using custom beans

In addition to the core AbleBeans provided in the ABLE component library, users can easily wrap new or existing algorithms to create their own beans. Sets of domain-specific beans can be added to the ABLE Agent Editor and dynamically loaded from a JAR (Java ARchive) file. A simple design pattern requires that the algorithm object be wrapped by an AbleBean instance, a BeanInfo file be created to specify any members to be externalized, and a GUI Customizer class be provided to allow users to set any algorithm unique attributes.

This approach was used to incorporate the DecisionTree and NaiveBayes classifier learning components into ABLE. As shown in Figure 7, the three Java classes required for ABLE integration are the AbleBean class itself, the BeanInfo that defines bean

Figure 7 AbleBean wrapper design pattern



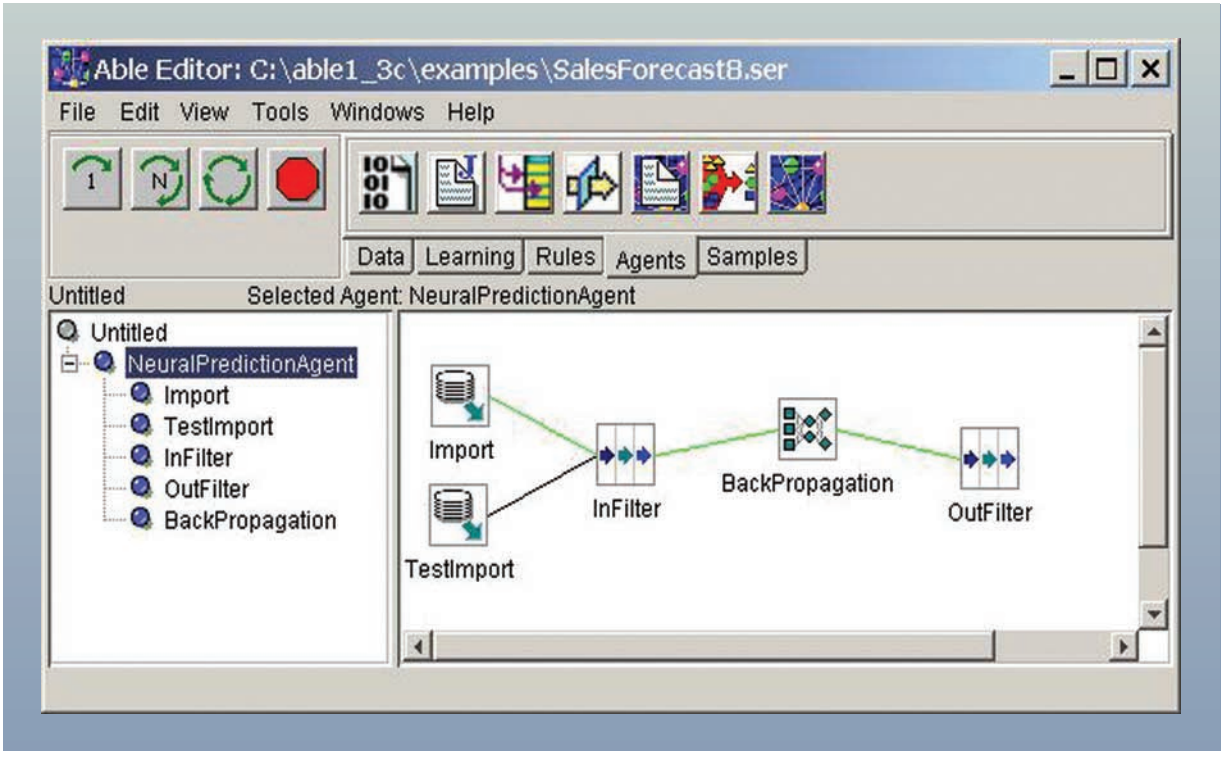

properties and accessor methods, and the bean Customizer that provides the GUI used to set configuration properties on the bean. The AbleBean must contain an instance of the algorithm class, map the init( ) and process( ) methods to call functionally equivalent methods on the algorithm object, and also wrap any getter and setter methods used by the Customizer. This approach allows the algorithm code to remain unmodified while allowing it to be used as part of any ABLE solution.

## ABLE development tools

The ABLE Agent Editor is a Swing-based interactive development and test environment. It provides a tree view of the agent with drill down into contained beans as well as a canvas view of the agent with corresponding data, event, and property connections. Agents can be loaded, edited, and saved to external files using Java serialization. ABLE Inspectors provide text and graphic views of object data using Java introspection. Support is provided for adding custom inspector views in addition to standard line, bar, *x-y* plot, and pie charts.

The Agent Editor can graphically construct AbleAgents by using the library of core AbleBeans and AbleAgents as building blocks. Data-flow, event, and property connections can be added using the GUI environment. Agents can also be hand-coded and then tested in the ABLE Agent Editor. Each AbleBean provides a Customizer dialog that is used to configure it and to set property values. Figure 8 shows the ABLE Agent Editor with a single NeuralPredictionAgent bean loaded into a default agent. The user

Figure 8 The ABLE Agent Editor




has drilled down into the NeuralPredictionAgent and is viewing the five AbleBeans it contains. These beans are displayed in the right-side canvas.

When the Agent Editor is started, it loads AbleBeans from JAR files. These beans contain properties specifying the page on the toolbar palette where the bean should be placed. Thus, users can easily provide their own custom AbleBeans and AbleAgents for use with the Agent Editor in combination with the core AbleBeans.

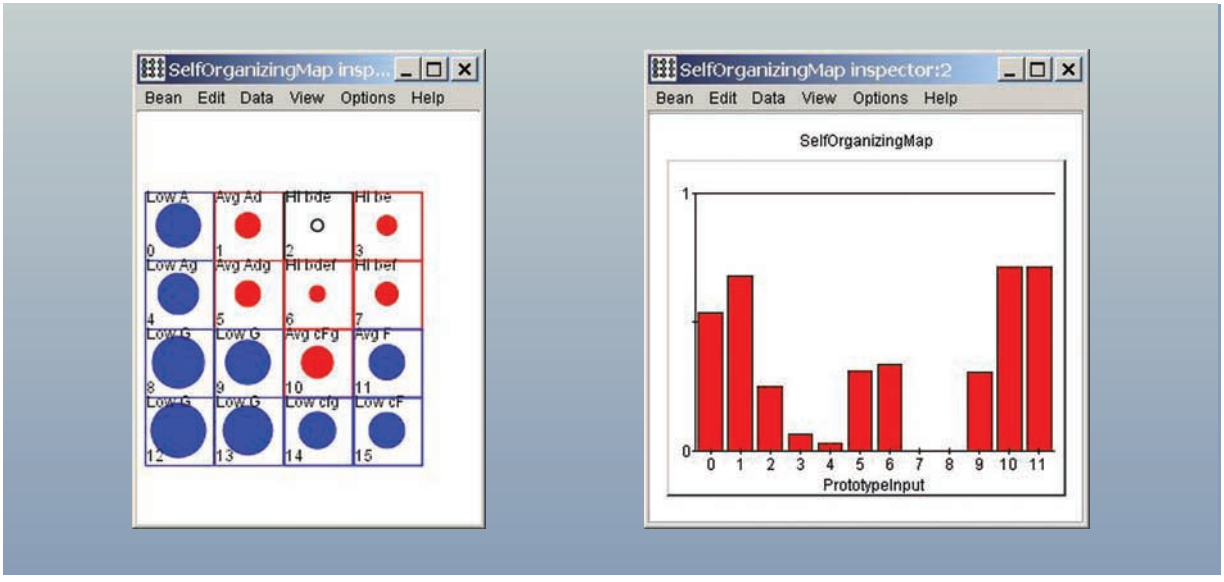
ABLE Inspector windows use introspection to display bean data members and state information. Inspectors provide text views as well as graphical views of the bean data. Views such as bar charts, line plots, and *x-y* plots are provided. Users can select one or more bean properties to be displayed, or one or more indexed properties such as arrays or vectors of objects. In addition, Inspectors allow users to select multiple data items for use in time-series displays. The Inspector caches the data points at each time step and displays the desired number of points. This function is very useful for observing time-series predictions and agent controller behavior over time.

Figure 9 shows two Inspectors from the MarketAnalysis example provided with the ABLE toolkit. The Inspector on the left shows the clusters of a SelfOrganizingMap neural network with labels and categories assigned to each cluster. The Inspector on the right shows a bar chart of the weights of the winning cluster.

## ABLE agent platform

The ABLE agent platform provides a set of services for AbleAgents that form multiagent systems. The services include standard agent life-cycle transitions (e.g., create, suspend, resume, quit) as well as directory facilitator and agent communication functions. The ABLE platform is a distributed agent platform supporting agents on multiple physical systems that communicate using Java Remote Method Invocation (RMI). The ABLE agents can communicate with one another using the mechanisms described ear-

Figure 9 Example ABLE Inspectors




lier (AbleEvents or direct method calls) or with other FIPA-compliant agents and agent platforms through the use of the FIPA agent communication language.

As shown in Figure 10, the ABLE distributed agent platform corresponds to the FIPA abstract architecture. The current ABLE platform conforms to the FIPA 97 specifications. Work is in progress to adapt it to the more recent FIPA abstract architecture and conform to the Java Agent Services (JAS) being developed under the Sun Microsystems Java Community Process JSR 87. The JAS provides a set of Java interfaces and a reference implementation of the platform services required for a distributed agent platform that complies with the FIPA abstract architecture. The ABLE platform includes additional functionality covering agent life-cycle management, service registration, and agent security. The following services are provided as part of the standard services supplied by the ABLE agent platform:

- *Booter and Service Root*—provides the startup and root services to agents that want to communicate with the agent platform services and agents running on the platform
- *Naming Services*—provides a unique name for each agent registered with the platform
- *Transport Services*—provides a mechanism for agents to communicate via multiple underlying communication transports including Java RMI and HyperText Transfer Protocol (HTTP)
- *Directory Services*—provides a means for agents to register descriptions of themselves to allow other agents to find them and to find other agents
- *Life-cycle Services*—provides a Factory service that allows new types of agents to be added to the platform and for an administrator to create/start/suspend/resume/stop agents running on the platform. Support is also provided to move agents from one system to another on the platform.

The agent Console provides a centralized graphical user interface for administrators to access the directory services and life-cycle services on the platform. Agents can be created, configured, and deployed using the Console. Additional systems can be added or removed from the agent platform using the Console. Directory services can be queried to find the status of individual agents or of collections of agents based on service attributes.

## ABLE application designs

The ABLE toolkit is quite flexible and can be used to add intelligence to applications in a variety of ways. One or more core AbleBean components could be used in an application to provide specific functions. Additional domain-specific AbleBeans such as novel

Figure 10 The ABLE distributed agent platform

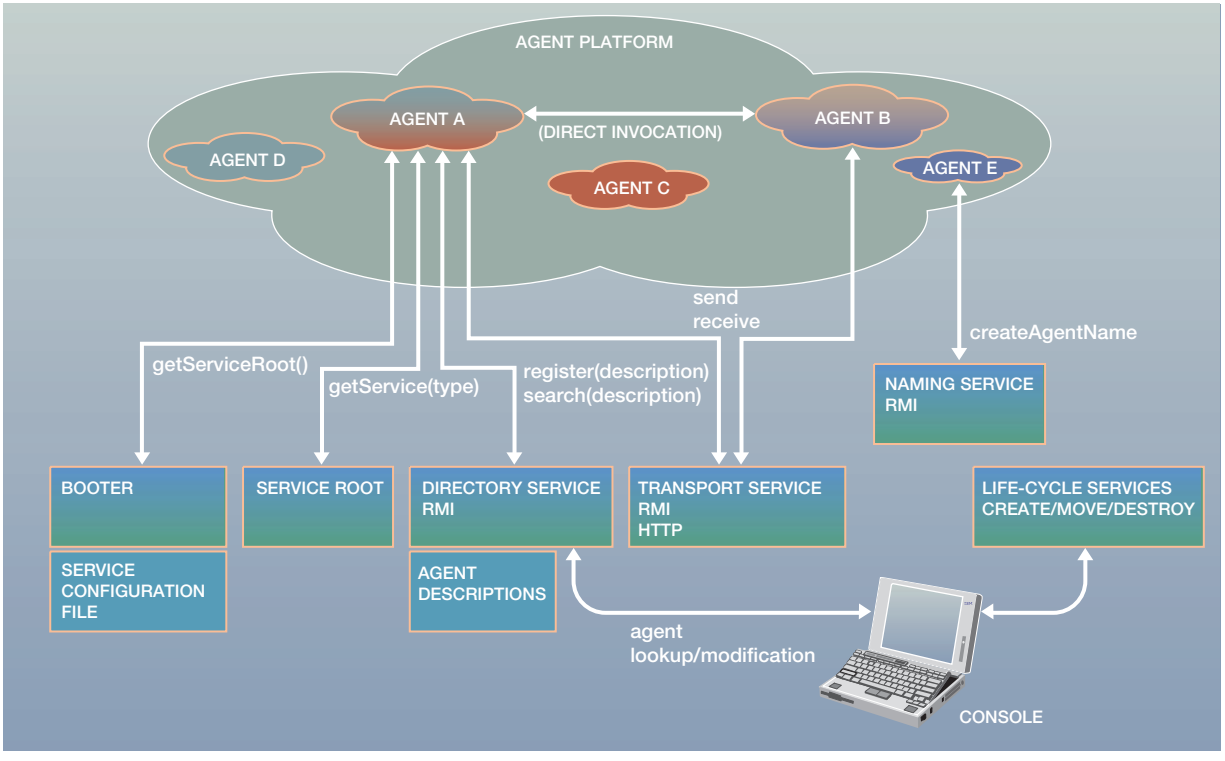
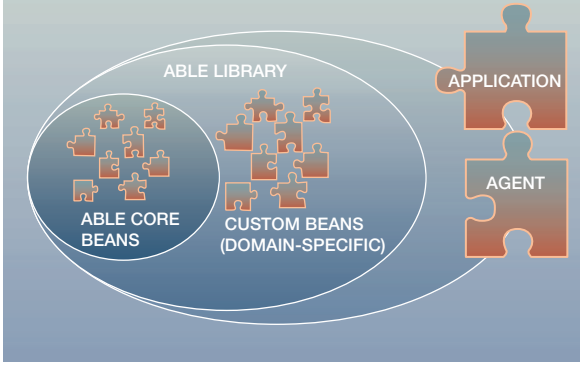



Figure 11 ABLE application design example

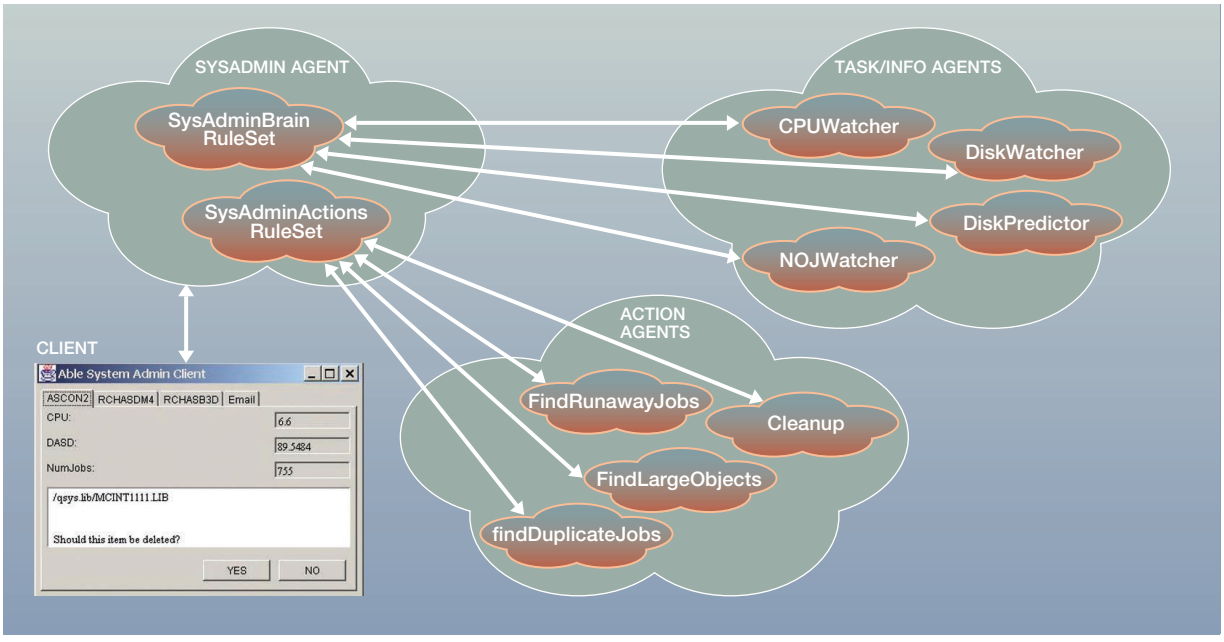



optimization or data processing techniques can be developed and mixed with the core AbleBeans. Autonomous agents composed of multiple AbleBeans could be used to provide function to the application. As illustrated in Figure 11, AbleAgents can bridge the gap between the ABLE component world and the application world by extending the AbleDefaultAgent and implementing the application-specific interfaces, or by extending a base application class and implementing the AbleBean interfaces. This approach allows all of the power of the AbleBean component library to be used to add new function to the application environment and is highly recommended.

In the following subsections, we present two application case studies that illustrate how ABLE components can be used to quickly develop solutions. The case studies are used as concrete examples of the various ways in which the ABLE framework, component library, development tools, and agent platform combine to enable development of multiagent autonomic applications.

**Case Study 1: System administration using ABLE.** A basic system administration multiagent system was developed for the IBM eServer iSeries* system using ABLE (see Figure 12). The goal was to provide an overall view of system health using multiple agents

Figure 12 System administration using ABLE




to monitor CPU utilization, the workload as indicated by the number of jobs running on the server, current disk utilization, and the expected disk utilization. The monitor agents are autonomous and use the built-in ABLE timer event processing to monitor the associated system resources at selected time intervals. When any one of the monitor agents detects a significant situation, it sends an event that is processed by the SysAdmin agent. When the SysAdmin agent receives the event, it is processed by an internal AbleRuleSet agent (called SysAdminBrain) that invokes other agents to gather additional information.

The SysAdminBrain AbleRuleSet can invoke one of the task agents to perform operations such as finding duplicate jobs, finding runaway jobs, finding large objects or files, and cleanup. The FindLargeObjects task agent uses the ABLE DBImport bean to perform a query to find the largest objects or files on the system. The Task agents are simply information gatherers. They do not take direct actions. The actions are performed by the SysAdminActions agent that contains another AbleRuleSet agent that either prompts the user to approve an action or automatically takes a remedial action such as killing a runaway job or deleting a system object.

A rudimentary SystemAdmin client GUI, shown in Figure 12, was developed using Java Swing. It communicates with the SysAdmin agent using the RMI connectivity that is built into the ABLE agent framework. The SysAdmin agent also sends a report of any actions it has taken to the client for display to the user. The client can also send these findings to a list of e-mail addresses so users can keep track of system management agent actions.

The SysAdmin agents were developed over a period of two weeks. The developers were new to the Java language and to the ABLE toolkit. They made use of the ABLE Agent Editor and AbleRuleSet editors to develop and test the agents and associated rulesets defining their behavior. They used the distributed RMI capability to build an application that runs on a single client system and can monitor multiple server systems. This case study demonstrates the productivity gains made possible through use of a standardized agent toolkit as well as the ingenuity of the developers.

**Case Study 2: A diagnostic application.** Another use of ABLE technology in IBM products is in the area of server diagnostics. The iSeries electronic support team is constructing a set of ABLE agents to perform

data collection, problem determination, and problem source identification tasks. The scenario is that a customer call comes into the support center. The customer support representative asks a series of questions describing the situation, symptoms, and other relevant information. One or more ABLE agents are then dispatched to the customer iSeries machine to help diagnose the problem.

Several steps are required to automate the diagnosis of machine problems. These steps include data collection, data formatting and preprocessing, data analysis and problem determination, and finally, advice or automation of the problem resolution. The ABLE toolkit provides beans and tooling to aid in all of these tasks. The ABLE Rule Language enables agents to call external programs to collect data. The AbleImports allow an agent to collect data and pass the data to another for analysis, using an external database or text file as the storage medium. An AbleDataTable bean provides a view over the external data in column major order. This view allows individual metrics to be analyzed as a time series and for groups of metrics to be correlated and analyzed in a time-series fashion.

Another example of a diagnostic application is an automotive diagnostic prototype developed in partnership with IBM Global Services. In this application, the ABLE Rule Language was used to perform time-series analysis over a 40-second time series, looking at multiple engine sensors to identify misfire conditions. In the production application, we foresee development of a comprehensive set of diagnostic agents, each capable of detecting the presence or absence of a particular fault or closely related set of faults. A diagnostic manager agent will coordinate the activities of the individual diagnostic agents, deriving the higher-order diagnosis (for example, three faults identified by individual agents are related and point to a single point of failure) and providing a diagnostic tree to find the root cause of the failure.

There are several advantages to the multiagent approach. Diagnostic agents can be developed incrementally to resolve the most common (or most difficult) problems first and, over time, can be combined to cover more and more of the problem space. The ABLE Rule Language can be used to define the diagnostic reasoning and data analysis tasks, providing additional flexibility. Agents can be developed by the support team and deployed at any point in the release cycle, as opposed to being tied into the system release schedules. For example, if an unexpected problem was found after the release of a new system, diagnostic agents could be made available to assist customers and support representatives at any time. Flexibility is one of the major advantages of an agent-based solution.

### Autotune agent

One of the basic operations required by autonomic systems is closed-loop control, where the state of a target system is monitored, compared to some desired goal state, and then adjusted as required to move toward the goal state.[14] As computer operating systems, middleware, and applications have become more complex; literally hundreds or thousands of parameters must be configured in order to keep everything running smoothly. A practical autonomic system would be composed of many controller agents, distributed across many computer systems, and operating at various levels in a hierarchy. Individual controller agents would receive high-level goals from above and, in turn, control resources and applications at lower levels in the hierarchy.

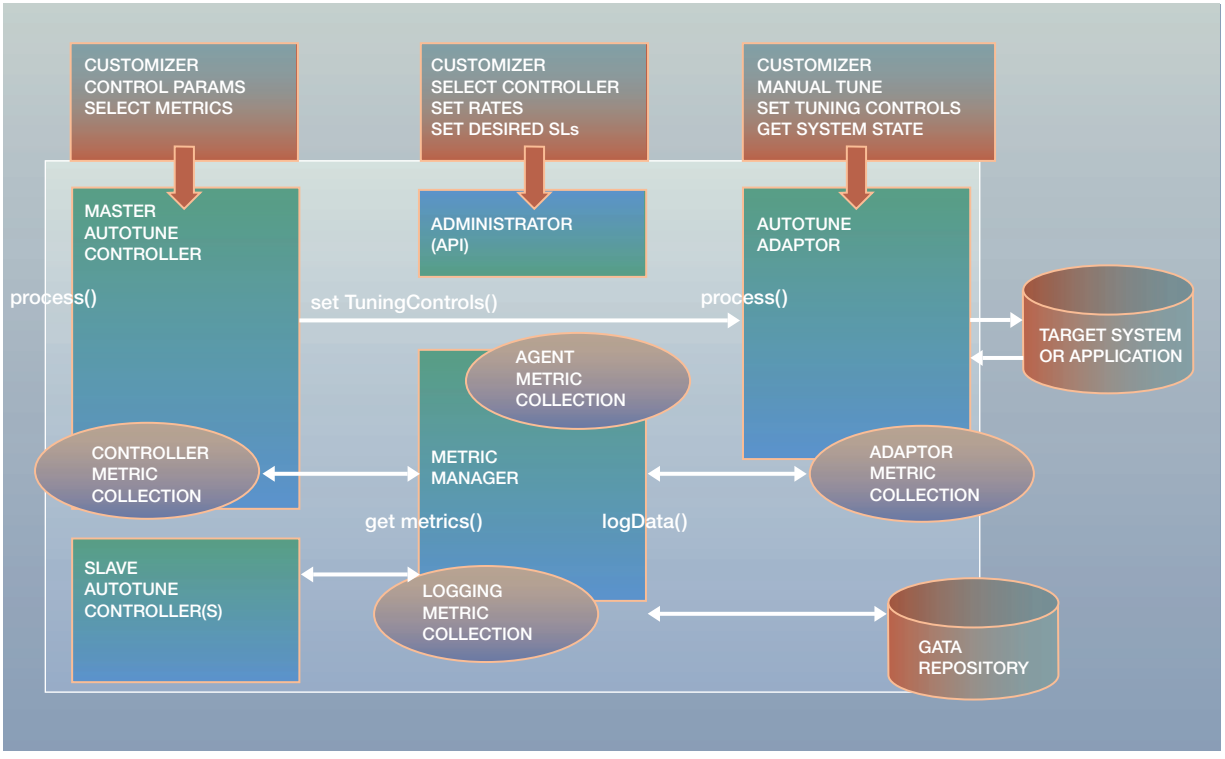
A generic AutotuneAgent has been developed that addresses many of these requirements. This agent extends the AbleDefaultAgent class but completely overrides the agent behavior. The Autotune agent contains one or more AutotuneController beans that provide control strategies and one or more AutotuneAdaptor beans to interface with target systems or applications.

A set of AutotuneMetric classes is defined to represent the state of the target system. Configuration-, Workload-, and Service-level indicators are read-only metrics that provide state information to the AutotuneAgent. TuningControl metrics can be dynamically set by the agent. The AutotuneAdaptor defines the set of metrics supported by a target system. These metrics are managed as a collection by the agent and can be logged to a historical data repository.

Figure 13 shows the architecture of an Autotune Agent. Each Controller bean provides its own Customizer GUI to allow the user to configure its parameters. Each Adaptor bean provides a data panel that allows the user to see and set target system metric values. The AutotuneAgent Customizers allow the user to select which Controller bean is the master (if there is more than one) and to set polling rate and other parameters.

The AutotuneAgent supports both distributed and hierarchical control: distributed by virtue of the agent

Figure 13 Autotune Agent architecture



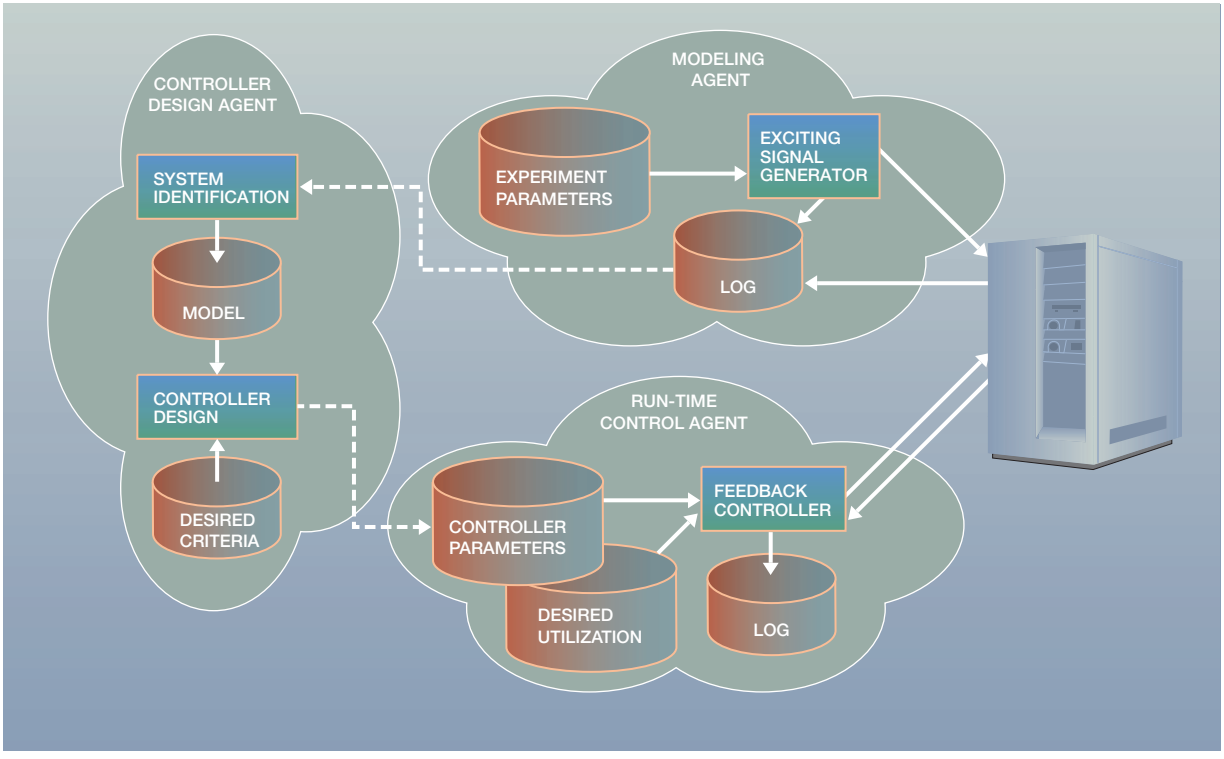

being an AbleBean, hierarchical when the target system is another Autotune agent. Autotune agents have been applied to tuning Lotus Notes* servers, Apache Web servers, and DB2* (DATABASE 2*) utilities.

## Case Study 3: An Autotune agent for Apache Web servers

In this application, a multiagent feedback control system based on ABLE Autotune agents was developed for automatically tuning the Apache Web server parameters. Typically, the Apache tuning work is done by the system administrator. The objective is to maintain the system CPU and memory utilization at a desired level so as to avoid overload or to reserve certain resources for other applications. This objective requires significant effort because the relationships between the desired CPU and memory utilization levels and the available tuning parameters (namely, MaxClients and KeepAlive timeout) are not clear.[15] Moreover, this tuning work must be done frequently since these relationships are affected by the workload, and the workload can vary over time.

To automate the Apache server tuning process, three Autotune agents were designed and built for the three phases in automatic feedback controller design and deployment (as shown in Figure 14). In order to understand the dynamic behavior of the server, a modeling agent is first applied to generate time-varying signals for the tuning parameters MaxClients and KeepAlive. Sine waves are used with magnitudes and frequencies specified through the Customizer GUI. The server behaviors (CPU and memory utilizations) under these exciting signals are recorded and passed to the controller design agent through text files and ABLE Imports. The controller design agent uses system identification techniques to extract a first-order linear model from the collected data. Based on this model, a linear quadratic regulation (LQR) controller is designed in order to meet certain design criteria specified by the user through the Customizer GUI, such as minimizing the difference between the desired and measured utilizations and minimizing the changes in the tuning parameters. The output of the controller design agent is a set of controller parameters that are passed to the run-time control agent. The desired utiliza-

Figure 14 Structure of the multiagent system for the Apache Web server




tion level is also specified as the control goal from the system administrator through the Customizer GUI. On the basis of this information, the feedback controller interacts with the Apache Web server to dynamically adjust the MaxClients and KeepAlive tuning parameters to meet the desired CPU and memory utilization levels.

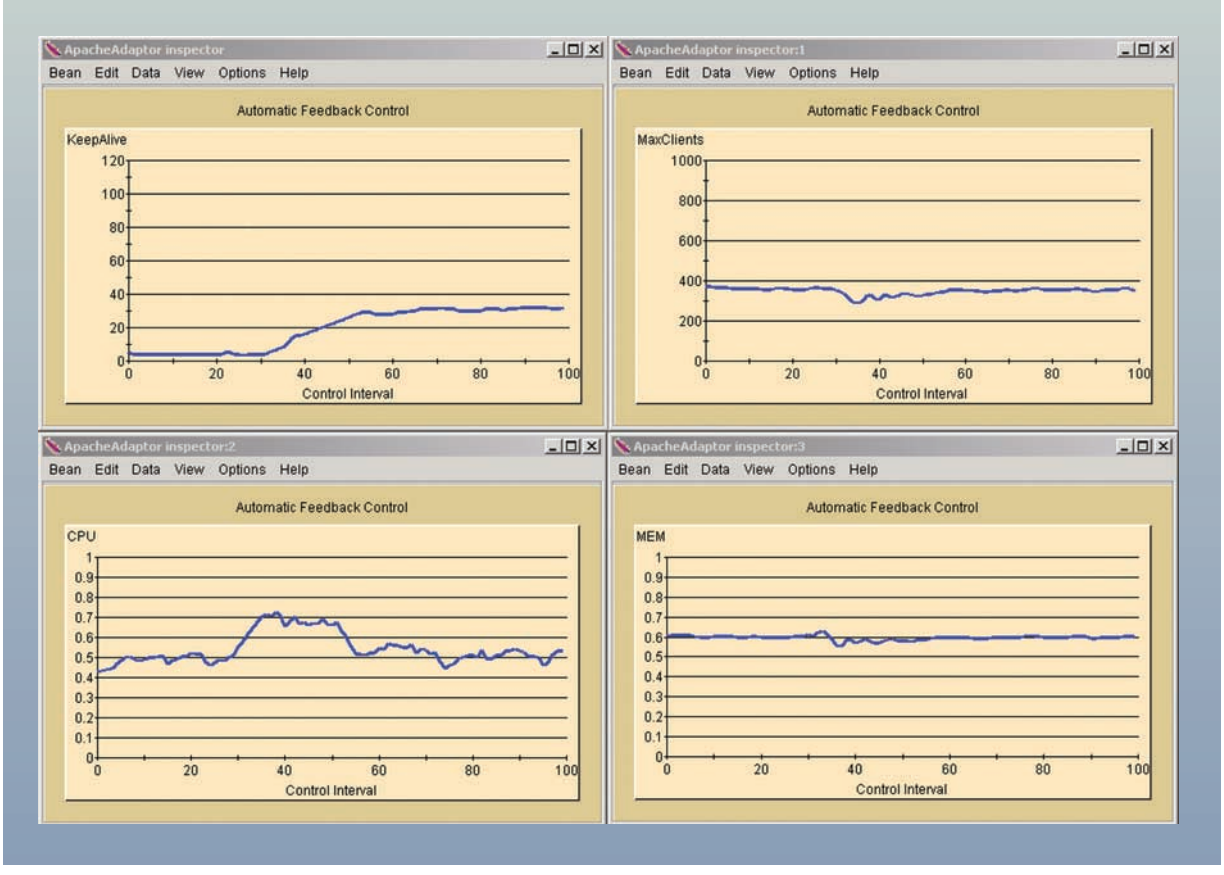
Figure 15 shows an example control run. The control interval is five seconds. Around the twentieth control interval the workload increases as a group of heavy users start to access dynamic Web pages (in contrast to normal users visiting static Web pages). This workload consumes more system resources, causes large increases in CPU utilization, and slightly increases memory utilization. In order to maintain the desired utilization levels (e.g., 0.5 for CPU and 0.6 for memory), the tuning parameters MaxClients and KeepAlive are automatically adjusted by the feedback controller. In particular, a larger KeepAlive value is used to decrease the CPU level, and the MaxClients value is adjusted temporarily according to the dynamics of the server.

## Subsumption agent

One of the basic tenets of artificial intelligence over the years has been the symbol system hypothesis, posited by Simon in 1969. His assertion is that people are intelligent because we process symbols and that only symbol-processing capabilities are required to produce intelligent machines. But this hypothesis begs the question of how those symbols become grounded to sensory inputs and perceptions from the real world.

In response to this problem, Rodney Brooks, who was working on robots at the Massachusetts Institute of Technology, proposed an architecture that relies on representations that are grounded in the physical world. Brooks says, "The key observation is that the world is its own best model." His subsumption architecture[16] was used to build a series of mechanical robots. This architecture uses the notion of layers of behaviors, each built upon the lower-level competencies and each responding directly to sensory inputs via effectors on the world.

Figure 15 Performance of the Autotune controller for the Apache Web server under dynamic workloads

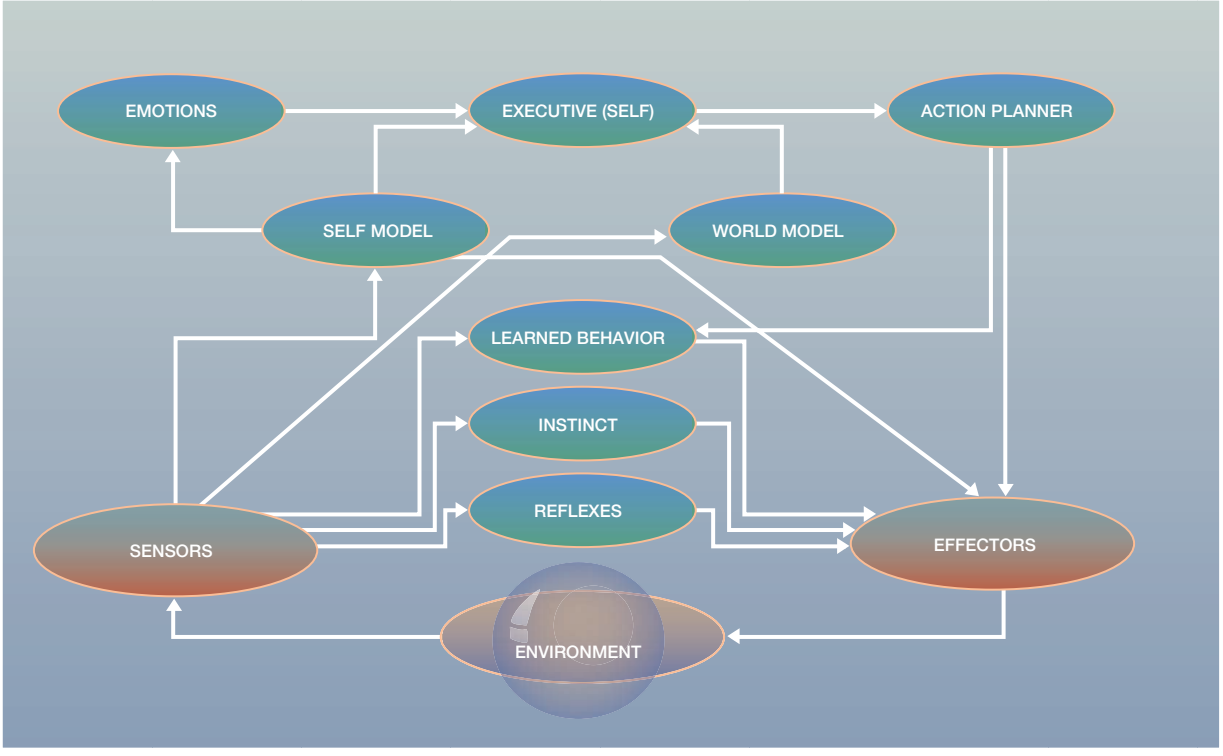



The AbleSubsumptionAgent extends the AbleDefaultAgent class and introduces the properties and behavior necessary to define new capabilities in such a system. There are three types of AbleSubsumptionAgent: reflexive, reactive, and adaptive. Reflexive agents are simple and fast. They respond quickly to changes in sensory input and usually use simple rules to define their behavior. Reactive agents are more complex. They take more time to evaluate and respond to changes in inputs, and often use data-driven forward inferencing or goal-directed backward inferencing. Adaptive agents learn from experience and modify their behavior based on past actions and subsequent feedback. All of these Subsumption agents contain the notion of levels or fixed priority as defined in the subsumption architecture. Although subsumption had a strict hierarchy, multiple AbleSubsumptionAgents can reside at the same priority level and represent alternative cooperative or competitive approaches in responding to inputs.

## Autonomic agent

In this section, we outline an architecture and methodology for building an autonomic agent capable of playing a role in a future autonomic computing infrastructure. The AbleAutonomicAgent extends the AbleDefaultAgent and contains multiple autonomous or semiautonomous AbleAgents from the ABLE component library. The internal agents cooperate and compete to take control of the intelligent system and make the appropriate response, much like Minsky's *The Society of Mind*. The base agent architecture is shown in Figure 16. The architecture contains a central pool of subsumption agents similar to that of Brooks[16] with sensors and effectors providing inputs from and outputs to the external world. Three defined behavior layers implement the basic reflexive behaviors, the complex instinctive reactive behaviors, and the more complex behaviors learned from interactions with the world. These three

Figure 16 An autonomic agent architecture




levels are implemented by AbleAgents that are containers for multiple AbleSubsumption behavioral agents that implement domain- or situation-specific behaviors.

Our intelligent autonomic agent must build and maintain a model of the external environment and of its own components. A top-level executive component makes decisions based on the models and its current emotional state. A planner component is used to create multiple step scripts or sequences of actions necessary to achieve the high-level goals being pursued by the executive. Like the behavior levels, the models, emotions, executive, and planner components are all implemented as AbleAgents that in turn may be composed of other AbleAgents and AbleBeans. The high-level architecture defines the data and event flows between the components in the autonomic system. By constructing the base agent using ABLE self-similar components, intelligent autonomic systems of widely varying complexity could be built using this architecture. The system retains all of the advantages of the reactive behavior-based subsumption architecture while adding internal mental states, including models of the self and world, emotions, learned behaviors, planning, and meta-level decision-making.

Our thinking has been influenced by prior work in this area, most notably Sloman and Minsky. Riecken has implemented the M system that corresponds to Minsky's architecture with multiple reasoning agents, blackboards for communications, rule-based inferencing, and semantic networks for representing domain objects.[17] Butler et al.[18] have implemented an object-oriented version of Brooks' subsumption architecture.[19] Sloman was one of the first to discuss computer-generated emotions, and his three-layered model with reactive, deliberative, and reflective processing levels is somewhat similar to this architecture.[3] Caulfield and Johnson sketch an architecture for a "conscious" system whose components correspond to the Autonomic agent architecture.[4] Jonker and Treur[20] present a multiagent architecture intended to simulate animal behavior. Picard[21] gives a good overview of the computational

issues related to machine generation and recognition of emotional states.

The novelty in our approach is the use of an agent structure with well-defined functional components, where those components themselves are multiagent systems. The ABLE framework and agent platform make constructing systems like this feasible. The existing ABLE component library provides us with a rich set of machine learning and reasoning capabilities on which to base our implementation. Although we have just started down this road, our experience with building other higher-level agents, such as the Autotune agent, gives us confidence. Our firm belief is that any truly autonomic system will require one or more agents of this type as part of the architecture.

## Concluding remarks

Our objective in this paper was twofold: to describe the set of functionality provided in the ABLE toolkit and to demonstrate its utility via real application case studies. Although we selected three examples, they are just a few cases where ABLE has been used. We have applied the ABLE agents to multiple problems in systems management, including event processing, performance monitoring using adaptive thresholds, system health monitoring using hierarchies of fuzzy rules, and time-series prediction for service-level agreement management using neural networks.

ABLE components have been successfully applied to e-commerce, including computing complex discounts in a business-to-business environment using IBM WebSphere* Commerce Suite. The ABLE rule engines have been used in conjunction with the BRBeans component in the WebSphere Application Server Enterprise Extensions. The ABLE framework and component library will be shipped as part of an upcoming iSeries operating system release. Application agents for performing communication traces and data collection are slated for production use by the iSeries eSupport organization. Additional systems management agents are also in development.

We continue to add new algorithm beans to the ABLE component library. The development of the AbleSubsumptionAgent and AbleAutonomicAgent will take place over the next year. We plan to add the IRIS (information, representation, inferencing, sharing) hypergraph knowledge representation to use as an integrated method for encoding and reasoning about domain knowledge.[22]

We have described a series of agents that could play the role of intelligent nodes in an autonomic computing system. One could easily imagine networks of distributed intelligent agents managing storage, operating systems, network resources, database and file systems, middleware, and applications while simultaneously being managed by other agents in the hierarchy. At various points in the network, we may require relatively simple agents, dominated by reflexive behaviors. At higher levels we may require complex reactive behaviors, learning, and adaptation. It is unlikely that we will reach a fully functional autonomic computing system in one giant leap. We will need to incrementally expand the depth and breadth of intelligent behaviors available to the individual agents as well as to the entire distributed autonomic system.

In the future, we plan to leverage our work on the ABLE toolkit to further explore the world of autonomic computing. This grand challenge will likely attract a large number of researchers using a wide variety of technical approaches. We intend to attack the autonomic computing problem from an agent-based perspective. We plan to build on this work by adding higher levels of abstraction and sophistication to our agents and our agent platform as we pursue the goal of building truly autonomic computing systems.

## Acknowledgments

The authors would like to acknowledge and thank the iSeries and eServer teams, including Kristi Schultz, Mike Smith, Bob Hansen, Jack Corcoran, Earl Emerick, Carol Egan, Kenton Lynne, Jason Illg, Mark Masbruch, and Tony Dunbar. Special thanks to Jake Kugel and Jeff Awe for their work on the System Administration agents.

We would like to thank the Tivoli and WebSphere teams, including Mark Johnson, Carole Corley, Kristof Kloeckner, Mark Linehan, and Rob Will.

We thank our collaborators at the IBM Thomas J. Watson Research Center, including Joseph Hellerstein, Sujay Parekh, T. S. Jayram, Mark Squillante, Irina Rish, Ricardo Vilalta, Michelle Zhou, Rosario Uceda-Sosa, and Isabelle Rouvellou.

## Cited references and notes


1. M. L. Minsky, *The Society of Mind*, Simon & Schuster, New York (1985).
2. R. A. Brooks, "Intelligence Without Representation," *Artificial Intelligence Journal* **47**, 139–159 (1991).
3. A. Sloman, "Damasio, Descartes, Alarms, and Meta-Management," *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics*, San Diego, CA (1998), pp. 2652–2657.
4. J. H. Caulfield and J. L. Johnson, "Softest Computer," *Proceedings of the Society for Optical Engineering*, Bellingham, WA (2000).
5. *Autonomic Computing: IBM's Perspective on the State of Information Technology*, IBM Corporation (2001), available at http://www.research.ibm.com/autonomic.
6. A. B. Damasio, *Descarte's Error: Emotion, Reason, and the Human Brain*, Putnam, New York (1994).
7. V. S. Ramachandran, *Phantoms in the Brain*, Morrow, New York (1998).
8. J. P. Bigus and J. Bigus, *Constructing Intelligent Agents Using Java*, Second Edition, John Wiley & Sons, New York (2001).
9. The Foundation for Intelligent Physical Agents, http://www.fipa.org.
10. J. P. Bigus, "The Agent Building and Learning Environment," *Proceedings of Autonomous Agents 2000*, Barcelona, ACM Press (2000), pp. 108–109.
11. J. P. Bigus and K. Goolsbey, "Integrating Neural Networks and Knowledge Based Systems in a Commercial Environment," *Proceedings of the International Joint Conference on Neural Networks*, Vol. 2, Washington, DC, IEEE Press (1990), pp. 463–466.
12. Agent Building and Learning Environment, http://www.alphaWorks.ibm.com/tech/able.
13. J. P. Bigus, *Data Mining with Neural Networks*, McGraw-Hill, Inc., New York (1996).
14. S. Parekh, N. Gandhi, J. Hellerstein, D. Tillbury, T. Jayram, and J. Bigus, "Using Control Theory to Achieve Service Level Objectives in Performance Management," *Proceedings of IFIP/IEEE International Symposium on Integrated Network Management*, Seattle, WA (May 2001).
15. Y. Diao, N. Gandhi, J. L. Hellerstein, S. Parekh, and D. Tilbury, "Using MIMO Feedback Control to Enforce Policies for Interrelated Metrics with Application to the Apache Web Server," *Proceedings of Network Operations and Management Symposium*, Florence, Italy (April 2002).
16. R. A. Brooks, "A Robust Layered Control System for a Mobile Robot," *IEEE Journal of Robotics and Automation* **RA-2**, 14–23 (April 1986).
17. D. Riecken, "M: An Architecture of Integrated Agents," *Communications of the ACM* **37**, No. 7, 107–116 (July 1994).
18. G. Butler, A. Gantchev, and P. Grogono, "Object-Oriented Design of the Subsumption Architecture," *Software—Practice and Experience* **31**, No. 9, 911–923 (July 2001).
19. R. A. Brooks, "Intelligence Without Reason," *Proceedings of the 12th International Joint Conference on Artificial Intelligence*, Sydney, Australia (1991), pp. 569–595.
20. C. M. Jonker and J. Treur, "Agent-Based Simulation of Animal Behavior," *Applied Intelligence* **15**, No. 2, 83–115 (September/October 2001).
21. R. W. Picard, *Affective Computing*, MIT Press, Cambridge, MA (1997).
22. R. Uceda-Sosa, "Proactive Information in Distributed Knowledge Environments," submitted for publication in 2002.


*Accepted for publication April 1, 2002.*

**Joseph P. Bigus** *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York, 10598 (electronic mail: bigus@us.ibm.com).* Dr. Bigus is a Senior Technical Staff Member at the Thomas J. Watson Research Center, where he is the project leader on the ABLE research project. He is a member of the IBM Academy of Technology and is an IBM Master Inventor, with over 20 U.S. patents. He was an architect of the IBM Neural Network Utility and Intelligent Miner™ for Data products. Dr. Bigus received his M.S. and Ph.D. degrees in computer science from Lehigh University and a B.S. in computer science from Villanova University. His current research interests include learning algorithms and intelligent agents, as well as multiagent teams and their applications to adaptive system modeling and control, data mining, and decision support.

**Don A. Schlosnagle** *IBM Rochester Custom Technology Center, 3605 Highway 52 North, Rochester, Minnesota 55901 (electronic mail: daschlos@us.ibm.com).* Mr. Schlosnagle is an advisory software engineer in the Rochester Custom Technology Center. He studied computational linguistics and expert systems at the former IBM Systems Research Institute and wrote, in Common Lisp, a PC-based natural language DB2 query program that was successfully marketed by IBM as a PRPQ. Mr. Schlosnagle has since worked on the Neural Network Utility and on Intelligent Miner for Data, where he contributed the fuzzy logic inference system for evaluating proposed neural network architectures. A greatly enhanced version of the fuzzy system found its way into ABLE. His current interest is in combining neural networks with fuzzy logic.

**Jeff R. Pilgrim** *IBM Rochester Custom Technology Center, 3605 Highway 52 North, Rochester, Minnesota 55901 (electronic mail: pilgrim@us.ibm.com).* Mr. Pilgrim is an advisory software engineer contributing to the Agent Building and Learning Environment. His prior development work includes Intelligent Miner for Data, Neural Network Utility, wide area wireless computing, iSeries Management Central, and configurators for rack-mounted systems such as the 9221. Mr. Pilgrim joined IBM in 1979 at the former IBM facility in Owego, New York, where he was responsible for forecasting workload for defense contracts. Previously an APL zealot, he is now a Java bigot. He received his M.S. in industrial engineering and operations research in 1980 from the Pennsylvania State University.

**W. Nathaniel Mills III** *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598 (electronic mail: wnm3@us.ibm.com).* Mr. Mills joined IBM Research in 1996 after running a successful consulting and software development business for 10 years. His initial work in systems management led to the development of WebSphere Studio AE Page Detailer—a tool to measure and display Web page download performance. He is now a Senior Technical Staff Member working with ABLE on various projects involving automated problem diagnosis and systems management. These projects focus on "after sales" product support or are related to the product families of WebSphere and Tivoli. In particular, Mr. Mills helps build ABLE-based applications for use as embedded "rules engines" to help analyze and mine data, to make real-time assessments of various application states to drive problem diagnosis, and to recommend corrective action. Mr. Mills graduated from Trinity College in 1979.

**Yixin Diao** *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598 (electronic mail: diao@us.ibm.com).* Dr. Diao is a postdoctoral Fellow at the Thomas J. Watson Research Center. He received his Ph.D. degree in electrical engineering from Ohio State University in 2000. He has conducted research on fault tolerant control system design using an adaptive fuzzy/neural approach, intelligent reasoning for robust fault diagnosis, and nonlinear dynamic system modeling with hierarchical learning structure and multivariate statistical methods. He joined IBM Research in 2001. His work involves increasing the adaptive and learning abilities of computing systems to unknown and time-varying environments by exploiting techniques from control theory, machine learning, and distributed agents. In particular, he is working with ABLE to develop generic adaptive agents for automated server tuning. Dr. Diao has authored around 20 technical papers, and his research interests include computer performance management, intelligent systems and control, adaptive systems, and stability analysis.

# OptimalGrid -- autonomic computing on the Grid

## Research prototype is middleware that optimizes performance of your grid application

Level: Intermediate

James H. Kaufman (kaufman@almaden.ibm.com), OptimalGrid researcher, IBM
Tobin J. Lehman (Toby) (toby@almaden.ibm.com), OptimalGrid researcher, IBM
Glenn Deen (glenn@almaden.ibm.com), OptimalGrid researcher, IBM
John Thomas (jthomas@cruzio.com), OptimalGrid researcher, IBM

June 2003

In this article, we introduce OptimalGrid, a research prototype from grid researchers at the IBM Almaden Research Center. OptimalGrid is middleware that aims to simplify creating and managing large-scale, connected, parallel grid applications. It optimizes performance and includes autonomic grid functionality. You don't need to be a grid infrastructure expert to use it. You supply the code that represents your basic problem algorithm, and OptimalGrid manages everything else -- problem partitioning, problem piece deployment, runtime management, dynamic level of parallelism, dynamic load balancing, and even system fault tolerance and recovery. The OptimalGrid system is designed to bring the immense potential of Grid computing easily within reach of developers who aren't grid infrastructure experts.

## Introduction

Recently, there has been a surge of interest in the area of Grid computing, a way to enlist large numbers of (usually) heterogeneous machines to work on a multipart problem. In order to take full advantage of the enormous opportunities presented by the advent of Grid computing, use of the grid must become simple. Developers should be able to create grid-enabled parallel applications without, themselves, becoming experts in grid or high performance computing. Also, grid applications (or even larger grid systems) should be able to reconfigure both themselves and the resources they use in response to dynamic changes in the grid environment.

Early grid systems, such as CONDOR or SETI@home, provided the functionality to perform remote execution of program pieces over tens, hundreds, thousands, or even millions of machines. While this provides an excellent foundation for Grid and distributed computing, more functionality is needed.

CONDOR and SETI@home are examples of systems that support independently parallel problems, where each problem piece can be computed independently. These systems lack the ability to manage connected problems where the problem pieces are inter-related. They do not provide for sophisticated management of the problem pieces, account for correlations between problem pieces, provide a representation of problem piece requirements, nor do they adapt the problem itself to dynamic changes in available computing resources.

OptimalGrid is an attempt to simplify the creation and management of connected parallel applications on the grid. It is not a toolkit; it is a self-contained middleware that provides a grid-enabled collaboration framework and problem-solving environment. It is a layer between the OGSI infrastructure provided by Globus and applications that require distribution of interconnected problems on a grid. OptimalGrid will run on almost any grid infrastructure, requiring only that a Java runtime be installed on networked machines. It is designed to optimize performance to make the most of an existing grid infrastructure. This article provides a high-level overview of the major OptimalGrid components and describes architecture for a general interest audience. A companion tutorial provides more detailed information for developers who want to understand how to use OptimalGrid.

## Motivation -- hiding complexity

The price/technology curve of desktop and server machines has made computing cycles plentiful and inexpensive, opening the door for practical Grid computing and various models for compute utilities and services. Grid computing, a way to enlist large numbers of (usually) heterogeneous machines to work on a multipart problem [1-4], will make available low-cost computational resources at a scale previously unimagined. Business and scientific applications that require either extensive computation or extensive computation resources, such large amounts of memory or disk, are now available in abundance. Historically, it has been very difficult to create large-scale, connected parallel applications, as that required special expertise and access to expensive resources. Creating such applications for the grid added even more complexity. Today, using a grid to solve these complex problems requires application developers to be aware of the grid infrastructure they intend to deploy on, and requires them to design a custom application that can adapt to the grid environment.

Realizing a vision of ubiquitous parallel computing on the Grid requires not only the establishment of standards such as OGSA (Open Grid Services Architecture) [12] and tools such as Globus [1], it also requires middleware that effectively hides the complexity of creating and deploying truly parallel grid applications, making it easy for application developers to build and run connected parallel problems. Success in making the grid easy to use could revolutionize a variety fields from manufacturing design to the science of medicine.

The simplest class applications to address with a computational grid have so far been too "independently parallel" (sometimes called "embarrassingly parallel") problems. The CONDOR system [5], from the University of Wisconsin-Madison, was one of the first systems to offer a remote execution service -- for either self-contained programs or for sets of parallel program pieces. The SETI@home project and the Folding@home protein folding project are also examples of such applications. These applications work in a simple "scatter/gather" mode and have no requirement for communication between the grid nodes participating in the computation. Such problems, while well suited for the distributed computing power of a grid, are straightforward to create. Such applications typically don't require or employ autonomic features to actively manage and maximize the effective use of available resources. Issues such as failed nodes or missing data sets can often be dealt with by simply rerunning a calculation. The lack of connections between the problem pieces allows such simple responses to failures since there is no affect on the calculations being performed on other nodes.

A much larger and more general class of applications can be described as "connected parallel" problems. These include finite element model problems (and a subset of these are cellular automata problems, such as The Game of Life)[8]. Finite Element Model (FEM) problems are common in the commercial world and are solved using a set of well-understood techniques. FEM problems can be found in a diverse set of expertise domains, including physics, financial systems, life sciences, and complex simulations. In these application classes, cells or individual problem pieces are connected to adjacent neighbors. It is necessary to communicate the state of all neighboring (connected) cells in order to calculate the future (next) state or properties of a cell.

In this paper we introduce OptimalGrid, a resesarch project at IBM Almaden Research Center. We designed OptimalGrid to simplify the creation and management of large-scale, connected, parallel applications in a grid environment. In order to simplify the creation of such connected parallel applications on a grid, we divide the problem into two steps. The first step involves partitioning the application and application data. Conceptually, a large-scale connected parallel application is an application that won't "fit on" or run effectively on one machine. It must be divided so as to make use of multiple machines. This is, in many cases, the most intellectually challenging step. The OptimalGrid middleware hides the complexity of this step in an "auto problem building" operation completed at load time. If one considers the parallel application as a graph where the nodes on the graph contain data, methods, and pointers to neighbors, the problem is to partition that graph into pieces of a convenient size to run on single grid nodes. This object model and our approach to problem partitioning are discussed in OptimalGrid object model and our approach to problem partitioning .

The second step in deploying a grid application involves managing the application at run time. To effectively use a Grid environment one must adapt to changes in the available Grid resources and dynamically load balance the application. This requires real time diagnostic and application performance measurements and feedback to a coordinator that can make decisions about how to manage the application. See more details in the section Autonomic features.

In Run-time optimization we briefly describe the runtime optimization steps used by OptimalGrid in solving a connected problem.

## OptimalGrid object model and our approach to problem partitioning

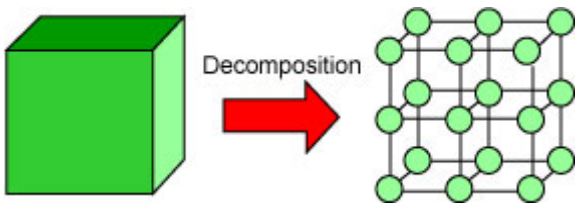
First, let's talk about the OptimalGrid object model and intercell communication.

### The OptimalGrid object model

OptimalGrid is designed to facilitate the creation of connected parallel applications for the grid. It is not restricted to the scientific engineering domain, nor to the solution of spatial problems.

To understand the OptimalGrid object model, it is useful to consider a problem class that can be addressed with Finite Element Modeling. Any problem in which discrete elements have defined connected relationships with their neighbors can be solved using this technique. For example, financial simulations that use interdependent economic elements can be built using an FEM approach. FEM models are not limited to two or three dimensions. Finite element model (FEM) problems are solved numerically by partitioning space into "small" finite regions or elements where "small" is typically defined by the smallest relevant length scale in a problem.Figure 1 illustrates a continuous solid object being modeled by a discrete set of nodes, each with specific properties. Depending upon the problem being addressed these smallest regions might represent atoms, crystallographic unit cells, or simply grains within a solid.

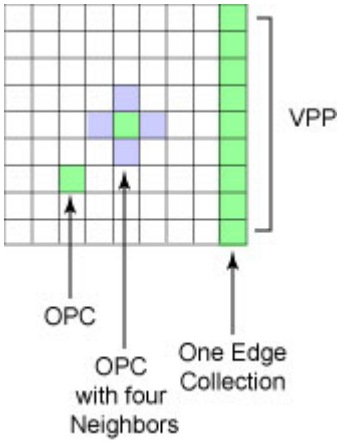
**Figure 1: Decomposing a solid object into finite elements**




To visualize the OptimalGrid system's approach, it is easiest to consider the simple two-dimensional problem, as shown in Figure 2. In this example we assume *nearest neighbor* connectivity so each element is connected to its four closest neighbors. With *periodic boundary conditions* edge elements

also have four neighbors (some at remote edges of the problem map). With non-periodic boundary conditions, edges have two or three neighbors depending on their location.

**Figure 2: Original Problem Cell**




We define the smallest piece of a problem (in this case, a single element) as an Original Problem Cell or OPC. In the abstract, an OPC represents a node on the *application graph* and contains data, methods, and pointers to other OPCs. The OptimalGrid object model implements code to solve a problem using abstract OPCs. The user implements a small set of methods defined or required by an OPC Abstract Class. These methods describe the connectivity of the cell with its neighbors, and specify the calculations to be performed by the cell using local data and information obtained (automatically) from connected neighbors. Typically, a single OPC object is very small, requiring little memory and minuscule computational power to execute. The OptimalGrid system aggregates collections or groups of OPCs that are connected to one another to form an *OPC collection*.

Once created, OPC collections are fixed in size for the lifetime of a problem. Load balancing is accomplished by exchanging OPC collections between grid nodes (each node handling one or more collections at a time). We chose this architecture over load balancing at the level of individual OPCs to avoid inefficient and excessive accounting overhead. Problems that require the computing power of a grid typically have very large numbers of OPCs. It would be impractical to optimize a grid of a thousand nodes, each with a million OPCs, if each OPC were tracked and managed individually. Of course, for special types of problems, an OPC collection could be defined as a collection of only one OPC.

**Definition:** A **map** is the description of the connections between elements.

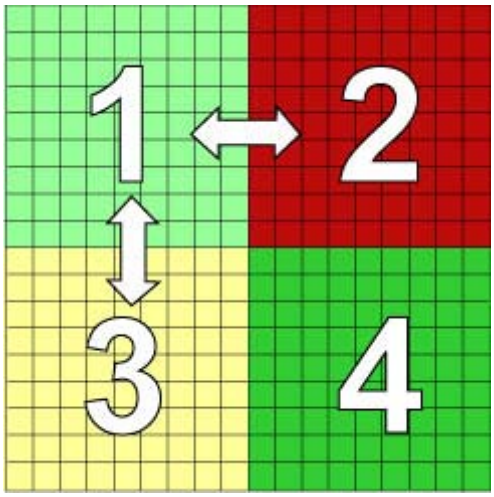
The problem partitioning step defines the OPC collections to make load balancing practical and efficient. The application is divided in such a way that the communication cost will not grow arbitrarily as OPC collections are later exchanged between nodes. One or more OPC collections define the "problem piece" assigned to a compute node. The problem piece object containing a set of OPC collections is defined as a Variable Problem Partition, or VPP.

**Definition**: A **Variable Problem Partition**, or VPP, is the set of OPC collections assigned to a grid node. The number of OPC collections contained in a VPP is variable.

## Intercell communication

As shown in Figure 3, each cell or OPC can communicate with multiple neighbors. Therefore, the OPC collections and the VPPs also share data with neighbors -- including neighbors processed on remote machines.

**Figure 3. Intercell communication**

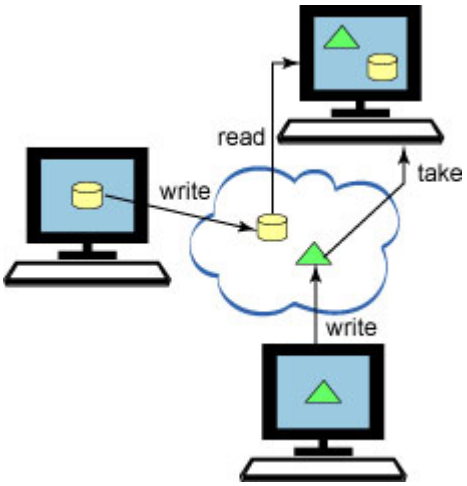



An OPC in a cellular problem of dimension N will typically need to communicate with a collection of dimensionality N-1 if the problem has "near neighbor" connectivity. As explained, OPCs are grouped together into OPC collections that are never changed in size. OPC collections, in turn, are dynamically assigned to a VPP for execution on a common grid node. The cells within an OPC collection communicate to one another directly using in-memory (Java object references) communication. The cells at the edge of an OPC collection will communicate with the edge of neighboring OPC collections using in-memory communication if they are on the same compute node (such as, in the same VPP). If the OPC collections are not within the same VPP, then data is communicated over the existing network infrastructure. The OptimalGrid system attempts to minimize the amount of network communication needed in solving a problem.

The OptimalGrid system employs a distributed whiteboard model in which clients have access to one or more shared global message boards, as opposed to using multiple point-to-point communication connections. This is a common element of Tuplespace communication systems [9] [10]. The OptimalGrid system uses TSpaces[11] that additionally provides a lightweight database system coupled with a Tuplespace communication system.

The example shown in Figure 4 is a trivial one, having only one whiteboard for all nodes. If all nodes used only one whiteboard, the system would have very limited scalability. Instead, a set of distributed whiteboards is used.

**Figure 4. Basic TupleSpace operations**




The optimal ratio of communications whiteboard nodes to VPP nodes depends on the actual problem solved. Factors such as the amount of data being reported during edge and sequence communications must be considered. This is a configuration step that is automated by another component of the OptimalGrid system: the Problem Builder. The Problem Builder is run at program load time (if the problem is not already partitioned). It is run when the problem is initially being assigned to nodes on the grid. The Problem Builder examines the problem and determines the optimal number of grid nodes needed to solve the problem; this plays an important role in self-optimization. Presumably, the application data might be so large it will not fit in the memory of a single machine. OptimalGrid auto-partitions to problem based on the following procedures:

1. Determines the complexity of the application instance (number and size of all OPCs)
2. Determines the number of grid nodes available
3. Uses algorithms based on known OptimalGrid performance and scaling to predict the optimal number of grid nodes to use

4. Optionally interacts with user to select division into optimal number of pieces defined in step 3, or to divide the problem into some other number of pieces (in which case OptimalGrid predicts the computation time). This step may later enforce execution per some service level agreement in an on-demand service model.
5. Creates a spatial filer that predefines the dependencies between empty OPC collections
6. The filter (which requires little memory) is applied to the application data (which may reside on a file system) to serially generate the OPC collection objects
7. The user may optionally apply a VPP data initializer class to customize the state of OPCs before launch of the application instance
8. Program launch

### Autonomic Program Manager

The OptimalGrid system employs a component known as the Autonomic Program Manager (APM). The APM function is assigned at runtime to one of the grid nodes allocated to the problem. Using the instrumented performance and diagnostic data generated by the VPPs, the APM acts as an application coordinator to direct and optimize VPPs. The APM decides on the dynamic assignment of OPC collections to VPPs. The use of distributed whiteboards makes this easy to implement, without adding either large amounts of costly communication or data management overhead. The APM is the brain behind the autonomic features in OptimalGrid. An integral part of the APMs design is the ability to add *pluggable* policy modules, allowing specific optimization goals to be introduced for a problem, such as Service Level Agreement (SLA) goals, performance goals, and recovery goals. The APM is not a grid coordinator. It does not attempt to administer the physical grid. One APM instance is created for every application instance and responds to changes in the grid environment to optimize the application within the bounds of the resources available for that particular application instance.

## Autonomic features

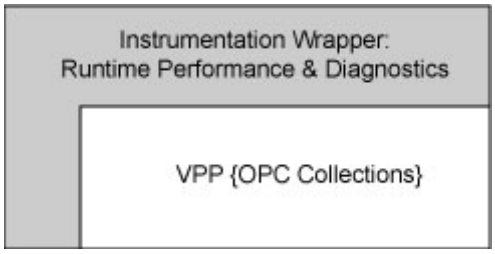
Orchestrating complex connected problems on heterogeneous distributed systems is not a job that even an expert administrator could manage. Thus, it's important that our system be as self-healing and self-organizing as possible. To this end, we have created a system with many pieces of instrumentation and a certain amount of knowledge, with rules on maintaining balanced performance and reacting to various types of failures. Over time, after running many experiments and dealing with our customers, we hope to augment this knowledge considerably.

### Instrumentation

Enabling autonomic function requires that system performance be measured and *fed back* to the APM. The OptimalGrid system is instrumented to allow for active management based upon real events. This allows OptimalGrid to make decisions based on both predicted activity, and actual

results in real time. A shown in Figure 5, the diagnostic and performance measurements are communicated in a lightweight *Instrumentation Wrapper* that can encapsulate messages communicated as part of the normal execution of the grid application.

**Figure 5. Instrumentation of VPP providing runtime performance data**




## Self-configuring

When the OptimalGrid system initializes itself to solve a problem, it retrieves from the grid a list of available compute nodes. Along with this list, it also obtains a set of normalized performance measurements from each computer node. This information either is read from an existing data file on the node, or is generated by running a set of simple self-tests on the node. The performance information, and the number of available nodes is passed to the Problem Builder which, using knowledge of characteristics of the problem to be solved, calculates an initial optimal assignment of the number of compute nodes, and the distribution of OPC collections to the VPP on each node.

The optimal number of grid nodes is not necessarily the number of available grid nodes. Grid nodes provide computational power and memory. Communications between all OPCs contained on a single grid node is performed in memory and is very fast. When a problem is extended across two or more nodes, the additional cost of network communication must be taken into account. Using algorithms that understand the problem map, and such factors as compute power, memory, and network latency, the Problem Builder creates a resource partitioning of the optimally needed grid nodes out of the available grid nodes. The goal of the Problem Builder is to devise *both* the *grid plan* and the *problem plan* that will result in *optimal* performance on an existing grid infrastructure.

## Self-optimizing

**Definition**: One round of calculating the value of all the OPCs in the problem is a **cycle**.

At the end of a cycle, the value of each OPC is communicated to all of the neighbors of that OPC. OPCs on the edges of a VPP (connected to remote grid nodes) communicate with OPCs in other VPPs via a whiteboard. In many applications, the actual time to compute a single cycle is quite short -- microseconds. To support recovery from failed nodes, it is necessary to record the value of all

OPCs in a collection periodically. However, the communication cost to do this at the end of every cycle, potentially every microsecond, would have an enormous impact on performance. To reduce the communications cost, only edges are communicated at the end of each cycle. The values of interior OPCs are communicated or *logged* at the end of a set of cycles, or a sequence. The number of cycles in a sequence is configurable allowing for the selection of the optimal length for a particular problem. Problems such as rendering movie frames require the writing of all OPCs at the end of every cycle, hence a sequence of length 1 cycle. Other problems, such as solving a connected series of equations which will in the end produce a single value for the entire problem, only require interior OPCs be logged at intervals short enough to allow for quick recovery from a failed node.

**Definition:** A **sequence** is a configurable number of cycles. At the end of a sequence the total state (values) of all OPCs in a VPP is written to a whiteboard.

The ability to configure the length of sequence provides the ability to optimize the amount of network communication and configure the logging of state in an appropriate way for different applications. The data stored on the whiteboards at the end of each cycle, and at the end of each sequence, is useful for recovery functions, as discussed in the section Self-healing .

## Computation and network optimization

The OptimalGrid middleware provides run-time performance measurements of each node with respect the particular VPP that node is handling. These measurements include both the computation cost, the communication cost, and the latency (obtained via timestamps added by the whiteboards). The performance measurements are encoded in the lightweight instrumentation wrapper and can be passed along with any messages being sent as required by the application. For example, diagnostic measurements are included in a simple clock tuple sent by each node to a whiteboard at the end of every cycle. Here the whiteboards play another important role. Because messages can be read (as opposed to just "taken"), messages on whiteboards are visible to multiple components and can even be broadcast. To dynamically load balance, the autonomic program manager (APM) or problem coordinator can monitor the performance via the data in the instrumentation wrapper. Consulting a pluggable rules engine, the APM can reassign VPPs between grid nodes, reassign OPC collections between VPPs, etc. These reassignments are made based on compute performance, network performance, or both. Ideally, to optimize the use of each grid node, each node should be fully utilized. Any time spent "waiting for a message" should be used for computation within the core of a VPP, and any message should arrive just as the grid nodes complete the core computation. After a sequence of execution, in which one or more iteration of interactions between OPCs has occurred, the OptimalGrid system evaluates the performance of each compute node, and then rebalances the

size of VPPs assigned to each compute node by reallocating the assignment of OPC collections to the VPPs.

### Self healing

The distributed whiteboards used by the OptimalGrid system enable the ability for the system to self heal if one or more compute nodes fail during a sequence. The whiteboards contain the historical values of the edge collections of OPCs from all the VPPs working on the problem. The loss of a grid node during a sequence does not result in the loss of the complete calculation thus far performed by the grid. Instead, all that is lost are the results of the interior OPCs inside the VPP on the failed compute node.

Upon detection of the failure of the compute node, the OPC collections in the failed VPP are reassigned to existing VPPs. Starting from the last VPP state recorded at the end of the last successful sequence, the node is able to play "catch up" and re-execute the OPCs in the VPP. During this catch-up phase, the VPP is able to use the edge results from its neighbors that were previously reported and stored. Thus, only the lost OPCs in the failed VPP need to be recalculated during the catch-up phase. This is possible because the required edges from neighbors are already known and available on the whiteboards, using their database feature.

Unavoidably, by nature of the connectedness of the problem, the other compute nodes must remain idle during this catch-up phase, but such a short delay is preferable to having to restart the problem solution from the beginning. The length of this delay is configurable by varying the sequence duration, allowing the application developer to trade off recovery time from failure against communication costs.

Once the lost OPC collections have been recalculated, the grid is again ready to continue working on the overall problem.

## Run-time optimization

When running a problem, OptimalGrid goes through these steps:

1. **First sequence**: When the cycles in the first sequence are completed, the VPPs write to their associated whiteboard all the OPCs they contain. The VPPs also write their performance instrumentation results to the whiteboards. The whiteboards now hold the final results of the sequence for all VPPs. The edge results from the cycles in a previous sequence can now be purged, as they are no longer needed for performing recovery of a failed node. This then completes the first sequence.
2. **Re-optimizing the problem**: At the end of the sequence, the Autonomic Program Manager evaluates the results of the sequence. Using the actual performance results encountered for

the sequence, it then redoes the assignment of OPC collections to VPPs. Grid nodes that took longer to complete their assigned task have some of their OPC collections removed from their VPP and assigned to those nodes that had better performance. The VPPs are then notified to being another cycle/sequence.

3. **Reacting to failure**: If at the end of a cycle a node fails to write its edge OPC collections to a whiteboard, the APM will detect this via a "clock" message. Different options are possible on when to take action to deal with this. The effect of the lost edges will propagate through the problem OPCs like a ripple moving one set of OPCs away from the failed edge per cycle. It is thus possible to continue processing for a number of cycles before the entire system must be paused to allow for reaction to the lost VPP. The loss of a VPP is dealt with by the APM in the recovery catch phase.
4. **Getting the results**: After the required number of sequences has been executed, the problem is complete. The data results reside on the whiteboards. This data is read and stored for the user to retrieve. The system then releases the allocated nodes back to the grid.

## Status and future plans

The OptimalGrid Project continues to be actively pursued at the IBM Almaden Research Center in San Jose. Future releases will include integrated support for the Open Grid Services Architecture (OGSA) and automatic deployment and configuration tools. For more detailed information on the use of OptimalGrid, please see the tutorial.

## Conclusion

The OptimalGrid system is designed to bring the immense potential of Grid computing easily within reach of users who are not grid infrastructure experts. By including autonomic features such as self-configuration, self-optimizing, and self-healing into its core architecture, OptimalGrid seeks to deliver a robust system capable of handling truly connected problems, meeting a broad class of user needs, and providing a high level of reliability.

---

## Resources

- Get more information about the Globus project at http://www.globus.org.
- The CONDOR system, from the University of Wisconsin-Madison, was one of the first systems to offer a remote execution service. Other examples of simple "scatter/gather" applications include SETI@home and Folding@Home.

- Finite Element Model (FEM) problems are common in the commercial world and are solved using a set of well-understood techniques. For examples, see Algor, Inc. at http://www.algor.com; Altair Engineering, Inc. at http://www.altair.com; Ansys, Inc. at http://www.ansys.com; COMSOL, Inc. at http://www.consol.com; and EDS at http://www.eds.com.

- For information about TupleSpace communication systems, see D. Gelemter and A.J. Bernstein, "Distributed Communication via Global Buffer," Proceedings of the ACM Principles of Distributed Computing Conference (1982) pp. 10-18. Also, D. Gelemter, "Generative Communication in Linda," TOPLAS 7, No.1, p80-112 (1985).

- The TSpaces home page is at http://www.almaden.ibm.com/cs/TSpaces

- Find out about the Open Grid Services Architecture (OGSA) at http://www.globus.org/ogsa.

## About the authors

James H. Kaufman is a Research Staff Member in the Distributed and Cluster Systems Department at the IBM Almaden Research Center. Dr. Kaufman received a B.A. in Physics from Cornell University and a Ph.D. in Physics from U.C.S.B. Dr. Kaufman has made contributions to several fields of research at IBM. He is a Fellow of the American Physical Society. His current research interests include Distributed Computing, Simulation and Modeling, and Grid Middleware. James Kaufman can be reached by e-mail at kaufman@almaden.ibm.com.

Tobin J. Lehman (Toby) joined the IBM Almaden Research Center in 1986, shortly after finishing his Ph.D. degree from the University of Wisconsin-Madison. Toby's research interests include server-based backup systems, object-relational database systems, large object management, memory-resident database systems, Tuplespace systems, and computing grids. Toby is currently working on an autonomic Grid computing infrastructure for solving very large connected problems on a heterogeneous collection of Internet machines. Toby Lehman can be reached by e-mail at toby@almaden.ibm.com.

Glenn Deen is a Senior Software Engineer and a member of the OptimalGrid research team at the IBM Almaden Research Center. Since joining IBM in 1989, he has been involved in security architecture and distributed computing. His current interests include XML and Grid computing. He can be reached at glenn@almaden.ibm.com.

John Thomas is a Java developer for IBM. He was previously one of the lead programmers for the IBM Almaden TSpaces project. Currently he is a member of the OptimalGrid project at the Almaden Research Center. He works out of his home in Santa Cruz and can be reached by e-mail at jthomas@cruzio.com.

# Similar or Related Researches

The realization of the autonomic computing vision will require the energy and resources of researchers and labs around the globe

## 1. Berkeley University of California: OceanStore

OceanStore is a global persistent data store designed to scale to billions of users. It provides a consistent, highly-available, and durable storage utility atop an infrastructure comprised of untrusted servers. Any computer can join the infrastructure -- users need only subscribe to a single OceanStore service provider, although they may consume storage and bandwidth from many different providers. John Kubiatowicz is a researcher at Berkeley exploring the space of Introspective Computing, namely systems which perform continuous, on-line adaptation. Applications include on-chip tolerance of flaky components and continuous optimization to adapt to server failures and denial of service attacks.

URL : http://oceanstore.cs.berkeley.edu/

## 2. Berkeley University of California: Recovery-Oriented Computing

The Recovery-Oriented Computing (ROC) project is a joint Berkeley/Stanford research project that is investigating novel techniques for building highly-dependable Internet services. ROC emphasized recovery from failures rather than failure-avoidance. This philosophy is motivated by the observation that even the most robust systems still occasionally encounter failures due to human operator error, transient or permanent hardware failure, and software anomalies resulting from software aging. David Patterson is a Professor in Computer Science at UC Berkeley working on the ROC project.

URL : http://roc.cs.berkeley.edu/

## 3. University of Bologna, Italy: Anthill project

Anthill is a framework built to support the design, implementation and evaluation of peer-to-peer (P2P) applications. P2P systems are characterized by decentralized control, large scale and extreme dynamism of their operating environment and can be seen as instances of Complex Adaptive Systems, typically found in biological and social sciences. Anthill exploits this analogy and advocates a methodology whereby the

desired application properties correspond to the ``emergent behavior'' of the underlying complex adaptive system. An Anthill system consists of a dynamic network of peer nodes; societies of adaptive agents (ants) travel through this network, interacting with nodes and cooperating with other agents in order to solve complex problems. Anthill can be used to construct different classes of P2P services that exhibit resilience, adaptation and self-organization properties.
URL : http://www.cs.unibo.it/projects/anthill

## 4. Duke University: Software Rejuvenation

Software rejuvenation is a proactive fault management technique aimed at cleaning up the systems internal state to prevent the occurrence of more severe crash failures in the future. It involves occasionally terminating an application or a system, cleaning its internal state and restarting it. Current methods of software rejuvenation include system restart, application restart (partial rejuvenation) and node/application failover (in a cluster system). Software rejuvenation is a cost-effective technique for dealing with software faults that include protection not only against hard failures, but against performance degradation as well. Duke University collaborated with IBM to develop the IBM Director Software Rejuvenation tool , which is currently a part of the eLiza project.
URL : http://www.software-rejuvenation.com/

## 5. Edinburgh University: Structuring Information

Structuring information is an essential consideration for a holistic approach to autonomic strategies. An autonomic data structure based on human memory has been successfully developed and prototyped. While it is an information structure, it still exhibits the key characteristics described for autonomic systems and could be ideally suited for organizing data in an autonomic architecture. Its simple, homogeneous, self-referential structure provides many advantages over exclusively relational or hierarchical structures.
URL : http://www.neuromation.com/

## 6. University of Freiburg, Germany: Multiagent Systems

The research in the multiagent systems group is centered around economic approaches to self-organized coordination of multiagent systems, which is conceptually

based on a non-equilibrium market approach of Hayek and Mises (the catallaxy). The application area has been a model supply chain, and the agents, representing small businesses within the supply chain, have been able to coordinate the system by applying the "invisible hand", without any central coordination instance. These agents had been equipped with a learning algorithm based on a decentralized evolutionary algorithm to find prices and to change the bargaining strategy. Another related project is the "catallactic coordination" of content distribution networks. This is a joint project with the technical University of Barcelona, Spain, in the "Future and Emerging Technologies" research program. This topic has some connections with Grid computing, especially economic coordination issues like in Darwin, Radar or Globus.

**7. International Solvay Institutes for Physics and Chemistry, Belgium: Immunocomputing**

The aim of the Immunocomputing project is to implement the principles of information processing by proteins and immune networks in a new kind of computing in order to solve specific complex problems while being protected from viruses, noise, errors and intrusions.

URL : http://solvayins.ulb.ac.be/fixed/ProjImmune.html

## 8. University of Maryland, Baltimore County: Systems with Autonomous, Dynamic and Adaptive Components

The Ebiquity research group is exploring the interactions between mobile/pervasive computing, multi-agent systems, artificial intelligence, and e-services. The goal is to create systems based on the cooperation of autonomous, dynamic and adaptive components which are located in the "vicinity" of one another. These systems will be composed of a collection of independently designed components that automatically become aware of each other, establish (wireless) communication, exchange information about their basic capabilities and requirements, discover and exchange APIs, and learn to cooperate effectively to accomplish their individual and collective goals.

URL : http://research.ebiquity.org/

## 9. University of Michigan: Controls group

The controls group at the University of Michigan is an interdisciplinary group of faculty and students in the College of Engineering interested in dynamic systems, controls, and optimization. Dawn Tilbury is an Associate Professor in the Mechanical Engineering department at the University of Michigan in Ann Arbor. Her research interests lie in the area of control systems, and she is a member of the Controls Group in the College of Engineering.
URL : http://www.engin.umich.edu/research/controls/

## 10. Monash University, Australia: Service Oriented Distributed Computing and Data Intensive Computing

A Grid scheduling system, called Nimrod-G, has been developed to provide tools and services for solving coarse-grain task farming style parameter sweep applications on resources distributed around. It supports deadline and budget based scheduling driven by computational economy. The resource broker/Grid scheduler has the ability to lease resources at runtime depending on their capability, cost, and availability. Scheduling experiments have been conducted on our World Wide Grid testbed. These software tools and technologies have been created for service oriented distributed computing.
URL : http://www.csse.monash.edu.au/~rajkumar/ecogrid/

## 11. University of Texas at Austin: Qualitative Reasoning

Qualitative Reasoning (QR) research is part of the Artificial Intelligence Lab and the Computer Science Department at the University of Texas at Austin. The QR research group is supervised by Professor Benjamin Kuipers. Research in this area includes spatial reasoning and intelligent robotics, access-limited logic for knowledge representation, qualitative reasoning about the physical world and qualitative reasoning about the physical world.
URL : http://www.cs.utexas.edu/users/qr/

## 12. University College London, England: Bio-inspired Approaches to Autonomous Configuration of Distributed Systems

Ian Marshall, a visiting Royal Society Industrial Fellow from BT, is working with Lionel Sacks on bio-inspired approaches to autonomous configuration of distributed systems (including a bacteria inspired approach). Next generation networks require new control

techniques to increase automation and deal with complexity. Active networks in particular will require the management and control systems to evolve extremely rapidly, since users will be continuously adding new applications, services and virtual configurations. In our research we are exploring novel ad-hoc distributed control algorithms and architectures derived from biological and geophysical systems and measurements of fabricated systems such as the WWW.

## Conclusion

We saw the concepts of autonomic computing and IBM's many research projects, solutions and various strategies for autonomic computing that is new feature of computing environment in this report. Autonomic computing will be one of standard in computing area and be applied in many parts of industry and human life.

Now, autonomic computing capabilities span the breadth of IBM's product line, including software, servers, storage, and personal systems, as well as services and systems management. IBM is working with several standards bodies, including the Internet Engineering Task Force, Distributed Management Task Force, and Global Grid Forum, to leverage existing standards and develop new standards where none exist. IBM sponsors a joint program with IBM Research, the Autonomic Computing Technology Institute, where IBM is developing the advanced technology for Autonomic Computing. And IBM has been working with other industry leaders to meet the grand challenge of autonomic computing.

IBM's autonomic computing efforts are all about freeing customers to focus more on their business and less on their IT infrastructure. IBM is leading the effort in autonomic computing and is delivering autonomic solutions today.

# Affect and machine design: Lessons for the development of autonomous machines

by D. A. Norman
A. Ortony
D. M. Russell

Human beings have evolved a rich and sophisticated set of processes for engaging with the world in which cognition and affect play two different but equally crucial roles. Cognition interprets and makes sense of the world. Affect evaluates and judges, modulating the operating parameters of cognition and giving a warning about possible dangers. The study of how these two systems work together provides guidance for the design of complex autonomous systems that must deal with a variety of tasks in a dynamic, often unpredictable, and sometimes hazardous environment.

Animals and humans have two distinct kinds of information processing mechanisms: *affect* and *cognition*. Cognitive mechanisms—mechanisms that interpret, understand, reflect upon, and remember things about the world—are reasonably well understood. But there is a second set of mechanisms, equally important and inseparable—the system of affect and emotion that rapidly evaluates events to provide an initial assessment of their valence or overall value with respect to the person: positive or negative, good or bad, safe or dangerous, hospitable or harmful, desirable or undesirable, and so on.

Although affect and cognition are conceptually and to some degree neuroanatomically distinct systems, from a functional perspective they are normally deeply intertwined. They are parallel processing systems that require one another for optimal functioning of the organism. There is some evidence[1] that people with neurological damage compromising their emotional (affective) systems become seriously limited in their ability to organize their day-to-day lives, even while appearing to perform normally on a battery of standardized cognitive tasks. They become ineffective actors in a complex world. Furthermore, psychologists and others interested in artificial intelligence have repeatedly urged that affect is essential for intelligent behavior[2] by altering goal priorities and generating interrupts (e.g., References 3–5).

This paper[6] is intended to start a discussion about how the study of affect in biological systems might contribute to the development of autonomous computer systems. We suspect that from a functional perspective, some of the evolutionary forces that presumably led to the emergence of affect in animals are likely to be relevant to the design of artificial systems. However, we view this paper as only setting the stage for further research, realizing full well that it raises many more questions than it answers.

## A model of affect and cognition: Three levels of behavior

In this section we outline the essence of our three-level theory of human behavior, a work that is still in progress,[7] after which we discuss how these ideas might be applied to the development of large computer systems or computational artifacts. The ideas




0018-8670/03/$5.00 © 2003 IBM

we discuss are still incomplete, and their implications for the design of computer systems still quite speculative. Nonetheless, we believe that even our skeleton, incomplete as it is, provides potential lessons for the design of systems that have a variety of tasks and goals, that must run unattended and autonomously, and that need high reliability. Indeed, consideration of the design constraints on autonomous robots was one of the driving forces that led to this work.[8-13]

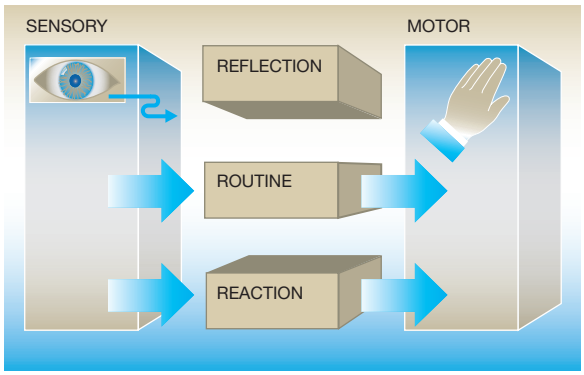
The three levels that we propose we refer to as the Reaction level, the Routine level, and the Reflection level (Figure 1). Processing at each level serves two different functions: evaluation of the world and what is happening in it—*affect*; and the interpretation of what is happening in the world—*cognition*. Higher levels involve greater depth of processing and concomitant slower processing. As shown in Figure 1, cognitive and affective information flows from level to level. Control information, in the form of activation or inhibition, flows downward.

**The lowest level: Reaction.** The Reaction level consists of the lowest-level processes. In animals, these processes are genetically determined and innate. No learning occurs. The Reaction level comprises immediate responses to state information coming from the sensory systems. Its function is rapid reaction to the current state.

The Reaction level monitors the current state of both the organism and the environment through fast, hard-wired detectors that require a minimum of processing. When it detects problematic or dangerous situations, it interrupts ongoing higher-level processing (if there is any), it heightens arousal, and it initiates an immediate response, or response preparation, along with a concomitant diversion of resources.

The output from the Reaction level is a set of fast and relatively simple interrupts, affective signals, and motor actions. Because of the rapid and relatively simple processing, the Reaction level cannot determine causes or do much more than respond in a simple pattern-directed manner. This level is the earliest of evolutionary processes, and in simple animals it is the only processing that occurs. In higher animals and humans, interrupts from the Reaction level trigger higher levels of processing (at the Routine and Reflection levels) in order to determine the cause and select an appropriate response. Responses at the Reaction level can be potentiated or inhibited by inputs from these higher levels, and they can habituate, reducing sensitivity to expected signals.

Figure 1 The three-level model




**The mid-level: Routine.** In humans, the Routine level is the level of skilled and well-learned, largely "routinized" behaviors. This level is the home of most motor skills, including language generation. The Routine level is quite complex, involving considerable processing to select and guide behavior. It must have access to both working and more permanent memory, as well as evaluative and planning mechanisms. Inputs to the Routine level come from the sensory systems, the Reaction level below, and the Reflection level above in the form of control signals (inhibition and activation). The Routine level can both inhibit and activate Reaction level responses and can pass affective information up to the Reflection level when confronted with discrepancies from norms or routine expectations.

The Routine level performs assessment, resulting in values on three dimensions, which are referred to in the scientific literature on affect and emotion as positive affect, negative affect, and (energetic) arousal.[14] Many emotion researchers now agree that positive and negative affect are essentially independent dimensions[15] as when the motivation of a person on a diet to devour a delicious-looking cookie (a source of positive affect) coexists with the motivation to avoid the same, fattening, cookie (a source of negative affect).

As alluded to above, a key feature of the Routine level is that of default expectations. When these expectations are not met, the system can make adjustments and learn. We return to this point later in our

discussion of possible applications. But note the power of expectations in signaling potential difficulties. In humans, these expectations trigger affective processes that play an important role at the higher level of processing.

**The highest level: Reflection.** Reflection is a metaprocess in which the mind deliberates about itself. That is, it performs operations upon its own internal representations of its experiences, of its physical embodiment (what Damasio[1] calls the "body image"), its current behavior, and the current environment, along with the outputs of planning, reasoning, and problem-solving. This level has input only from lower levels and neither receives direct sensory input nor is capable of direct control of behavior. However, interrupts from lower levels can direct and redirect Reflection-level processing.

**We suggest that affect can improve overall systems behavior.**

There is some evidence that affect changes the processing mode for cognition. The mechanism is neurochemical stimulation that adjusts the weights and thresholds that govern the operating characteristics of the cognitive mechanisms, biasing them and changing the nature of the ongoing processing. These changes influence how higher-level processing takes place, the locus of attention, and the allocation of attentional resources. Thus, negative affect, especially when accompanied by high arousal, appears to lead to more focused and deep processing—depth-first processing. In the extreme case, this type of processing leads to the "tunnel vision" of stress. In contrast, positive affect appears to lead to broad, more widely spread processing—breadth-first processing. As a result, humans have enhanced creativity when in a pleasurable state.[16,17] Both changes are, on average, evolutionarily adaptive (one being consistent with increased vigilance, the other with increased curiosity), even if at times they are counterproductive.

Note that we propose that Reflection has only indirect control (mediated through inhibition and activation) over behavior emanating from the Routine level. The mechanisms of this control have been explored more fully by Norman and Shallice.[18]

## Implications for machine design

Our artificial systems today have something akin to the three different levels of Reaction, Routine (action), and Reflection, but they do not distinguish between affect (evaluation) and cognition (understanding). In this section we discuss how a model of affect and cognition along the lines of the one we have proposed might apply to machines. Specifically, we suggest that affect can improve overall systems behavior, particularly in complex or difficult environments.

**The Reaction level in machines.** Reaction is the home of built-in sensors, usually with prewired or preprogrammed, fixed responses. This level is necessary for safety and other critical considerations for which a rapid response is essential. The Reaction level is essential to machine operation, and indeed, is already pretty well recognized and implemented. It is common for computer systems to monitor power and temperature, hardware functioning, and checksums. In robots and other mobile systems, Reaction-level devices include contact sensors and cliff detectors that prevent the devices from hitting other objects or falling down stairs.

In animals, when dangerous conditions are noticed, not only are higher levels of processing notified, but ongoing behavior is often altered. These alterations are generally very simply implemented, and the conditions for their elicitation are easily recognized. Machines can profit even from this elementary level of adaptation to important changes in their operating environments and, as indicated above, some do.

**The Routine level in machines.** The Routine level is the locus of routine computational work and so involves considerable computation and reference to prior events (memory). This activity differs markedly from analyses at the Reaction level. Thus, the detection of commonplace viruses and intruders requires analysis at the Routine level. (As viruses and intruders become increasingly sophisticated, it is more likely that their detection and the corresponding remedial actions will have to be initiated at the Reflection level.)

A key feature of humans and animals is the ability to respond to deviations from norms. Consider the value for computers were they to have some mechanism for recognizing such deviations. Suppose that

as programs traversed checkpoints, they were able to detect deviations from reasonable resource and time demands and that the detection of such a deviation would trigger an alarm. In this way, excessive time (or failure) to reach a checkpoint or the use of excessive resources would trigger a search for causes and possible termination of the program. Similarly, too fast an execution or too little use of resources would signal deviant operations. We believe that capabilities of this kind would greatly enhance the reliability and dependability of our computational artifacts. These capabilities are likely to be particularly important for autonomous robots.

**The Reflection level in machines.** The Reflection level is the level at which the system continually monitors its own operations.[19] This is both the highest level of analysis and the weakest in today's systems. Perhaps the most prevalent use of reflection is in systems that monitor such system behavior as load balance and thrashing. Reflection could lead to restructuring queues, priorities, or resource allocation. Similarly, detection of errant programs usually requires analyses at the level of Reflection. Once again, however, the automatic generation of cautionary behavior or even termination or avoidance of critical jobs does not seem to be common. Autonomous systems must have the flexibility to stop programs that could potentially lead to harm, that use excessive resources, or that appear to be in a deadlock.

***Example: redundant array of independent disks (RAID).*** Although RAID architectures are designed to offer robust, fast access to data stored in disk arrays, along with high reliability, data are still lost. Quite often loss results from the attempt to service a disk failure.[20] In theory, a disk failure should do no harm, since RAID arrays are designed to handle this contingency: the failed drive is pulled out and a good one put in. But occasionally the operator swaps out the wrong one, causing a second failure, and so data are lost.

There are a couple of approaches available to reduce data loss. One would be to make the RAID safe, even with two failures (e.g., RAID-6). A second would be to design the interface better to minimize such errors. This approach is clearly better: the value of efficient human-computer interaction is well-known, albeit too-seldom practiced. But the first approach comes at a price, namely, increased cost and loss of efficiency. Here is where the affective system would be useful.

Suppose that the loss of a disk drive is detected at the Reaction level and used to trigger an alert: in essence, the system would become "anxious." Yes, the human operator would be summoned, but here

**Lack of warning is a common problem in automated systems.**

the Routine level would kick in, retrieving past instances where service by the human operator had led to increased problems: this would serve to increase the anxiety level. The result of this increased anxiety would lead to an operations change—to a more conservative approach implemented by a change in policies. Because the margin of safety has been lowered, the system could institute more frequent checkpoint saves, perhaps to a remote location (after all, the RAID is no longer fully trustworthy), and perhaps the system could run a parallel shadow operation or postpone critical jobs. An alternative operation would be to restructure the RAID on the fly to make it tolerate further disk failure without damage, even at the cost of decreasing its capacity or slowing its operation.

In other words, why should computer systems not be able to behave like humans who have become anxious? They would be cautious even while attempting to remove the cause. With humans, behavior becomes more focused; they tend to engage in in-depth problem-solving first until the cause and an appropriate response are determined. Whatever the response for machine systems, some change in normal behavior is required.

Lack of warning is actually a common problem in automated systems.[21] The systems are well-designed to function even in the case of component failure, but they seldom report these failures to higher-level systems or change their behavior. As a result, the human operator, or higher-level monitors of the system, may be unaware that any problems have occurred even though error tolerance is now much reduced. Occasionally, further failures carry the system over the threshold of recoverability, often leaving the human operator to cope with the resulting unexpected emergency.

If systems followed the human model of affect, all failures would be reported, and just as in a person, a rising level of anxiety would trigger a change in focus and behavior at higher levels, preparing for eventual disaster and thereby minimizing its impact or possibly avoiding it altogether.

**Why use affect? Why not just program the system to safeguard itself against problems?** For any specific problem that might arise, once that problem is known and understood, the most effective solution will always be to write an appropriate algorithm to deal with it. So why are we proposing the introduction of a new system, that of affect? Why not simply analyze each potential failure and deal with it efficiently?

Normally, when thinking about computer systems design, we think in terms of what in artificial intelligence are referred to as *strong methods*, that is, methods that exploit specific domain knowledge and structure. In other words, we think in terms of specific algorithms that solve specific problems by incorporating substantial knowledge about the problem into the algorithm. By contrast, *weak methods* and *heuristics* do not incorporate domain knowledge because they are designed to be much more general. The result is that they are generally much slower, much less efficient, and often are not guaranteed to succeed. Weak methods trade efficiency for generality. Thus, for example, hill climbing is a weak method that has great generality, but is often inefficient and can become trapped by local maxima.

Strong methods are always preferable when the situations are known and understood and the environment predictable and relatively limited in scope. But when these conditions do not hold, weak methods are preferable. Affect is a computationally weak method. Its power lies in its capacity to help deal with unexpected problems, so that it complements strong, algorithmic methods by adding robustness in unanticipated situations. The real world is characterized by uncertainty and variability. For these cases, biology uses weak methods—methods that are general and applicable to a wide variety of situations. As machines become more autonomous and more exposed to uncertainty, affect will become an increasingly appropriate solution for them as well.

Biology, of course, is not without its strong methods. Even humans with their big brains have retained numerous wired-in, efficient responses to particular situations. Reflexes and tropisms respond rapidly to particular stimulus conditions such as lack of support, unbalance, bitter taste, the smell of putrefaction, and hot or sharp surfaces. These responses are rapid, pattern-driven solutions to specific classes of events. But biology also uses more complex, slower, reflective problem-solving and planning to deal with novel situations. Thus, biological systems make rapid responses to situations that require them, and slow, considered responses when circumstances demand them and time permits.

## Implications

An affective computer would be able to sense the state of its own operations and that of its environment. It would be able to compare its behavior with its expectations, and it would be able to reflect upon its own operations. It would have knowledge about its own trustworthiness and about that of the other systems with which it interacts, and it would be able to modulate its overall behavior toward better performance by sensing things that are not now taken into account, acting cautiously where appropriate and aggressively where possible. It would automatically reconfigure itself to take account of increased risk and would continuously be aware of the state of its own health, at least from an infrastructure and computational point of view.

We propose that by continually sensing its own state and that of its environment, the system would essentially be controlling its level of satisfaction or anxiety. When components needed service, the level of anxiety would rise, for the need for service means that error tolerances are lowered and the very act of service can cause errors. Just as human operators know not to do system maintenance or a software upgrade during or just before some critical job needs to be performed, so computer systems themselves should have the same sense of anxiety.

Imagine a grid computer, assembling a number of machines prior to doing a computation. Suppose that each machine were queried about its state of readiness, in essence asking "How are you feeling?" The range of possible responses given below is instructive:

"I had a disk failure in my RAID, so if this is an important calculation, you had better not count on me."

"I am feeling a bit anxious because I have had some errors, so I will be slowed by the need to do con-

tinual checks." (This response shows how a machine might provide a graded level of service.)

"I am feeling anxious because of recent virus or hacker attacks."

Animals have developed sophisticated mechanisms for surviving in an unpredictable, dynamic world, coupling the appraisals and evaluations of affect to methods for modulating the overall system. The result is increased robustness and error tolerance. Designers of computer systems might profit from their example.

## Acknowledgments

We thank Bill Revelle, Ian Horswill, and Tony Tang of Northwestern University for their contributions to the general theory and to the particular recommendations made here.

## Cited references and notes

1. A. R. Damasio, *Descartes' Error: Emotion, Reason, and the Human Brain*, G. P. Putnam, New York (1994).
2. M. Minsky, *The Emotion Machine*, Pantheon, New York, forthcoming.
3. N. Frijda and J. Swagerman, "Can Computers Feel? Theory and Design of an Emotional System," *Cognition & Emotion* **1**, No. 3, 235–257 (1987).
4. H. A. Simon, "Motivational and Emotional Controls of Cognition," *Psychological Review* **74**, 29–39 (1967).
5. A. Sloman and M. Croucher, "Why Robots Will Have Emotions," *Proceedings of the Seventh International Conference on Artificial Intelligence* (1981).
6. Originally presented at the IBM Autonomic Computing Summit at the Thomas J. Watson Research Center, May 14–15, 2002.
7. D. A. Norman, A. Ortony, and W. Revelle, "Effective Functioning: A Three Level Model of Affect, Behavior, and Cognition," in *Who Needs Emotions? The Brain Meets the Machine*, J. M. Fellous and M. A. Arbib, Editors, to be published.
8. There is beginning to emerge a substantial body of literature [9–12] on "affective computing," systems designed to recognize or simulate human affect. Much of this work, however, does not deal with the design of machine architectures. Our own views have been particularly influenced by work that does, particularly that of Aaron Sloman. [11,13] In common with Sloman, we propose that human information processing operates at three levels. Our three levels, the Reaction, the Routine, and the Reflection levels are related to, but somewhat different from those of Sloman (see Reference 13). In particular, whereas his reactive level is essentially the same as our Reaction level, his "deliberative reasoning" is related to but different from our "Routine" level, and his "meta-management" level is similarly related to but somewhat different from our Reflection level. Other differences are not relevant to this discussion.
9. C. Breazeal, *Designing Sociable Robots*, MIT Press, Cambridge, MA (2002).
10. R. W. Picard, *Affective Computing*, MIT Press, Cambridge, MA (1997).
11. A. Sloman and B. Logan, "Evolvable Architectures for Human-Like Minds," *Affective Minds*, G. Hatano, N. Okada, and H. Tanabe, Editors, Elsevier, Amsterdam (2000), pp. 169–181.
12. R. Trappl, P. Petta, and S. Payr, *Emotions in Humans and Artifacts*, MIT Press, Cambridge, MA (2003).
13. A. Sloman, "How Many Separately Evolved Emotional Beasties Live within Us?," *Emotions in Humans and Artifacts*, R. Trappl, P. Petta, and S. Payr, Editors, MIT Press, Cambridge, MA (2003).
14. R. E. Thayer, *The Biopsychology of Mood and Arousal*, Oxford University Press, New York (1991).
15. J. T. Cacioppo and W. L. Gardner, "Emotion," *Annual Review of Psychology* **50**, 191–214 (1999).
16. F. G. Ashby, A. M. Isen, and A. U. Turken, "A Neuropsychological Theory of Positive Affect and Its Influence on Cognition," *Psychological Review* **106**, No. 3, 529–550 (1999).
17. A. M. Isen, "Positive Affect and Decision Making," *Handbook of Emotions*, M. Lewis and J. M. Haviland, Editors, Guilford, New York (1993), pp. 261–277.
18. D. A. Norman and T. Shallice, "Attention to Action: Willed and Automatic Control of Behavior," *Consciousness and Self Regulation: Advances in Research*, Vol. IV, R. J. Davidson, G. E. Schwartz, and D. Shapiro, Editors, Plenum Press, New York (1986).
19. Our use of "reflection" is related to the sense intended in computational reflection, whether in programming languages or operating systems. Both uses emphasize the capability of a system to examine its own operations, but the details and goals differ.
20. A. Brown and D. A. Patterson, "To Err Is Human," *Proceedings of the First Workshop on Evaluating and Architecting System Dependability* (*EASY '01*), Göteborg, Sweden (July 2001).
21. D. A. Norman, "The 'Problem' of Automation: Inappropriate Feedback and Interaction, Not 'Over-Automation,' " *Human Factors in Hazardous Situations*, D. E. Broadbent, A. Baddeley, and J. T. Reason, Editors, Oxford University Press, Oxford (1990), pp. 585–593.

*Accepted for publication September 26, 2002.*

**Donald A. Norman** *Department of Computer Science, Northwestern University, 1890 Maple Avenue, Evanston, Illinois 60201 (electronic mail: norman@northwestern.edu).* Dr. Norman is Professor of Computer Science at Northwestern University and cofounder of the Nielsen Norman group where he serves on the advisory boards of numerous firms. He is also Professor Emeritus of Cognitive Science and Psychology at the University of California, San Diego. Dr. Norman is the author of numerous books, including *Design of Everyday Things* and *The Invisible Computer*. He has served as Vice President of Advanced Technology at Apple Computer, Inc. He has B.S. and M.S. degrees in electrical engineering and a Ph.D. degree in experimental psychology.

**Andrew Ortony** *Department of Psychology and the School of Education and Social Policy, Northwestern University, 2120 Campus Drive, Evanston, Illinois 60201 (electronic mail: ortony@northwestern.edu).* Prof. Ortony is a professor of psychology and education, and codirector of Northwestern University's Center for the Study of Cognition, Emotion, and Emotional Disorders. His primary research interests concern the relation between emotion, cognition, behavior, and personality, and the implications

of these for artificial intelligence (AI) and for interface design. His coauthored book *The Cognitive Structure of Emotions* is a widely used resource in AI applications involving emotions. Prof. Ortony is a Fellow of the American Psychological Association, the American Psychological Society, and a member of the American Association for Artificial Intelligence, the Cognitive Science Society, and the International Society for Research on Emotion.

**Daniel M. Russell** *IBM Research Division, Almaden Research Center, 650 Harry Road, San Jose, California 95120 (electronic mail: daniel2@us.ibm.com).* Dr. Russell is the senior manager of the User Sciences and Experience Research (USER) lab at the Almaden Research Center. The main interests of the laboratory are in the areas of designing the complete user experience of computation, especially in the domains of highly sensed or attentive environments, formalizing the characteristics of human behaviors for input mechanisms, and creating new ways of placing computation into the work space. Prior to coming to IBM, Dr. Russell worked at Xerox PARC and in the Advanced Technology Group at Apple Computer, Inc. He founded and managed the User Experience Research (UER) groups at both companies. He received his B.S. in computer science from the University of California, Irvine, in 1977 and his M.S. and Ph.D. degrees from the University of Rochester in 1979 and 1984, respectively.

# Autonomic computing 2: Implications for IT services

By Peter Andrews


## Summary

Autonomic computing has vast implications for information technology (IT) services. Its technologies are already being applied to business recovery and continuity. Outsourcing firms are predicted to be the primary market for autonomic computing because of the potential for flexibility and cost savings. Services personnel -- including architects, consultants and practitioners -- will need to adapt to include people in the system designs more effectively and to provide improved ease-of-use, knowledge management and security features for the purpose of meeting the requirements for hiding complexity from the user. Ultimately, autonomic computing promises to automate many of the assessment, integration, configuration and deployment tasks performed by IT services today. As this automation increases, design, planning and architecture will become even more critical, albeit at a higher, conceptual level.


## Background

The theory behind autonomic computing is to create systems that self-regulate, self-repair and respond to changing conditions. Based on this fact, it may appear to be a contradiction to say that IT services will play a crucial role. After all, one of the big selling points for taking on the challenge of developing autonomic computing is the reduction in labor costs. But one of the most attractive reasons for improving IT services is to make the technical capabilities of systems valuable to people and businesses. In fact, one of the first uses of autonomic computing is in the realm of disaster recovery and business continuity, and many of the characteristics that help define autonomic computing depend on the focused and creative work of architects, consultants, practitioners and other services personnel for their success.

## Characteristics of autonomic computing

Services will be needed to provide the characteristics of autonomic computing. The paper, "Autonomic computing: IBM's perspective on the state of information technology," lists the characteristics of autonomic computing. For instance, one characteristic is "an autonomic computing system must configure and reconfigure itself under varying and unpredictable conditions." A prerequisite for this characteristic is an abundance of modular software and a strict adherence to standards. In addition, there is also a requirement for careful planning of the systems architecture. Information design, particularly taking full advantage of Extensible Markup Language (XML), is needed so that new combinations are efficient and make sense. Just as important as the software and standards, a thorough analysis and understanding of human factors and the possibilities for

personalization will be critical. For example, many companies have redesigned their sites to allow significant customization by visitors. The return on that investment theory is that when users can personalize sites, they are more inclined to pay for subscriptions, use online bill payment services and promote products to friends.

Another characteristic of autonomic computing is that "an autonomic computing system never settles for the status quo -- it always looks for ways to optimize its workings." If that is true, then people need to be considered as part of the overall system. Real optimization must be considered in terms of current human needs, not just in terms of algorithms. This means that, from time to time, experts will need to be identified and -- though they might take advantage of decision support -- their analysis of the situation and its opportunities and challenges will be essential. The underpinnings of this decision support and analysis are human-centered knowledge management systems. These systems will need special attention paid to training, roles and ease-of-use. Given the variations in communities, this area of optimization will require the efforts of consultants who understand both tools and human motivations.

A third characteristic of autonomic computing is that "an autonomic computing system must be an expert in self-protection." While the spirit of this statement can be true, no system can deal with wily humans by itself. The world of security is particularly dynamic, with new approaches to intrusion being developed and harmful code being created constantly.

While immune systems and pattern recognition can help deal with novel situations, ultimately, the system must include human judgment because the trade-offs are rarely clear cut. People must weigh the risks, balancing the opportunities of openness against the potential threats. People also must analyze the situation to understand what is at stake and take responsibility for their choices. Services personnel need to provide environmental assessments, training and customized rules and policies. In short, they need to do everything from surveying business opportunities to thinking like a hacker -- the types of tasks that can not be handled entirely by systems.

Consider this next characteristic… "An autonomic computing system knows its environment and the context surrounding its activity and acts accordingly." If a computer system can know its environment, and more importantly, its context, it has the potential to significantly increase its value to us. The practical application of some of this -- such as optimizing storage capacity due to changes in the distribution of users -- is primarily technical. But consider how the value to users may extend if data is adjusted for a new interface in a way that is appropriate to each individual. It may include prioritizing information so that the messages that are the most important to users are presented in a way that is most likely to attract their attention. Collaborative filtering, ease-of-use, personalization and affective computing are all important tools for helping to realize higher value, and they all rely on making value judgments, including valuable insights on the preferences of individuals and useful assessments of their responses. People are integral to the context.

Finally, "an autonomic computing system will anticipate the optimized resources needed while keeping its complexity hidden." This characteristic is not only highly dependent on taking user needs into account, it must anticipate those needs. In order to anticipate user needs; user preferences, authorization limits and patterns of use must be understood. In addition, the system designer must understand and anticipate the many ways people interact with each other. Attention to social computing is particularly important when the system is called upon to support or even participate in online groups -- commonly called communities. People can belong to a variety of communities. Examples include: practitioners engaged in similar lines of work; employees across various functions that support a common product line; or students from different schools all interested in chemistry. All of this requires intense evaluation of, and sensitivity to, differences between individuals and communities. Even as the needs of the individual are detected, they must be balanced against the actual or potential needs of the community as a whole. For instance, optimizing solely based on whoever is first in line is not a good model.

## The services industry will be challenged

If the full vision of autonomic computing is to be realized, the IT services industry will need to evolve. If the system becomes more modular and standardized, lower-level tasks in deployment and configuration will be handled by the system. This can make people available for more challenging and interesting tasks. If systems are able to improve their reliability and to handle dramatic changes in capacity requirements automatically, the challenge will become building new types of IT services contracts and learning to manage service levels differently.

Probably the greatest benefit for companies that take advantage of autonomic computing will be realized in improved dealings with very large and dynamic systems that are designed to accommodate people of differing skill levels, priorities and goals, as well as those that vary in type and rate of change. Planning, strategy and applied imagination will be at a premium. A deeper understanding of the value, uses and support of community within a more complex technical environment will be in demand. Those employees involved with helping to develop autonomic computing systems will need to be expert listeners who can adapt to new situations and find creative ways to satisfy the needs and demands of users.

Here are some considerations for the future:

Who will be involved? For individuals and businesses alike, this will mean a greater dependence on teaming.

Who is responsible? With a much larger, more complex and more connected system, someone will need to take responsibility for glitches when they occur.

Who knows the answer? Increased complexity will require that people in different organizations work together to solve problems, so both the tools and the environment for knowledge management will need to

be strong.

How can an avalanche of new options created by autonomic computing and inventive uses for it be realized? Those companies that create environments that set the standard for being flexible and creative will be the winners in this new world. Social challenges, rather than technical ones, may be the most daunting elements of the "Grand Challenge" of autonomic computing.

## References

Various. "Autonomic computing: IBM's perspective on the state of information technology." Retrieved from the World Wide Web, March 5, 2002.
http://www.research.ibm.com/autonomic/manifesto/

## Other sites of interest

**Affective computing**

http://www.ibm.com/services/insights/etr_affective.html

**Autonomic computing** http://www.ibm.com/services/insights/etr_autonomic.html

**Collaborative filtering**

http://www.ibm.com/services/insights/etr_collaborative.html

**Communities of practice**

http://www.ibm.com/services/insights/communities.html

**O'Reilly and Associates, Inc.**

http://www.xml.com/

**Personalization**

http://www.ibm.com/services/insights/etr_personalization.html

**Pervasive computing**

http://www.ibm.com/services/insights/etr_instant_messaging_lessons_in_pervasive_computing.html

## Tek to watch

- Open source
- XML
- Encryption
- Agents
- Sensors
- Grid computing
- Simulation
- Pervasive computing

## About this publication

*Executive Tek Report* is a monthly publication intended as a heads-up on emerging technologies and business ideas. All the technological initiatives covered in *Executive Tek Report* have been extensively analyzed using a proprietary IBM methodology. This involves not only rating the technologies based on their functions and maturity, but also doing quantitative analysis of the social, user and business factors that are just as important to its ultimate adoption. From these data, the timing and importance of emerging technologies are determined. Barriers to adoption and hidden value are often revealed, and what is learned is viewed within the context of five technical themes that are driving change:

**Knowledge Management**: capturing a company's collective expertise wherever it resides -- in databases, paper, people's minds -- and distributing it to where it can produce the big payoffs

**Pervasive Computing**: combining communications technologies and an array of computing devices (including PDAs, laptops, pagers and servers) to allow users continual access to the data, communications and information services

**Real time**: "a sense of ultra-compressed time and foreshortened horizons, [a result of technology] compressing to zero the time it takes to get and use information, to learn, to make decisions, to initiate action, to deploy resources, to innovate" (Regis McKenna, *Real Time*, Harvard Business School Publishing, 1997.)

**Ease-of-Use**: using user-centric design to make the experience with IT intuitive, less painful and possibly fun

**Deep Computing**: uses unprecedented processing power, advanced software and sophisticated algorithms to solve complex problems and derive knowledge from vast amounts of data

This analysis is used to form the explanations, projections and discussions in each *Executive Tek Report* issue so that you not only find out *what* technologies are emerging, but *how* and *why* they'll make a difference to your business. If you would like to explore how IBM can help **you** take advantage of these new concepts and ideas, please contact us at insights@us.ibm.com. To browse through other resources for business executives, please visit: ibm.com/services/insights

Highly responsive computing systems with lower costs, greater agility and stronger resiliency

# Autonomic computing and IBM.

*Freeing companies to focus on their business—instead of on their infrastructure.*

**Systems complexity drives up costs**

While IT infrastructures are increasingly more complex and more costly to manage, the demand for uninterrupted information access intensifies. This need for information will only accelerate—even as businesses worldwide remain concerned over decreasing IT budgets and increasing demand for qualified IT professionals. And, considering the 24/7 nature of e-business, outages are not only expensive but also embarrassing.

Much of the complexity is caused by the proliferation of vendors and heterogeneous technology environments. Managing such environments entails that the components of a given solution be integrated and customized to fit into your business processes. Further complicating matters is the increased need to distribute data, applications and system resources across geographic and business boundaries. Given this additional complexity, the costs of managing the IT infrastructure—deploying, tuning, fixing and securing—can be quite high.

**Leading the evolution to smarter computing technologies**

What if your IT infrastructure could be more like you? As complex as it is, your body's autonomic nervous system has an amazing ability to perform a myriad of critical functions with no conscious effort or thought on your part. It regulates your heartbeat, adjusts your pupils to the light and controls your digestive system. This ensures that you have the energy to run when you need to, can easily step from the bright outdoors into a darkened theater and can maintain a healthy body.

Imagine the possibilities if your IT infrastructure could regulate their normal, internal functions as easily.

**Introducing autonomic computing**

Autonomic computing environment have the ability to manage themselves and dynamically adapt to change in accordance with business policies and objectives. Self-managing environments can perform such activities based on situations they observe or sense in the IT environment—rather than requiring IT professionals to initiate the tasks. Your IT professionals can then focus on higher value projects while the technology handles the more mundane tasks.

**Autonomic computing: technology now for the next generation of computing**

Environments with self-managing components can reduce the costs of ownership and operation. They possess intelligent and collaborative characteristics that can decrease costs and enhance your organization's ability to react to change.

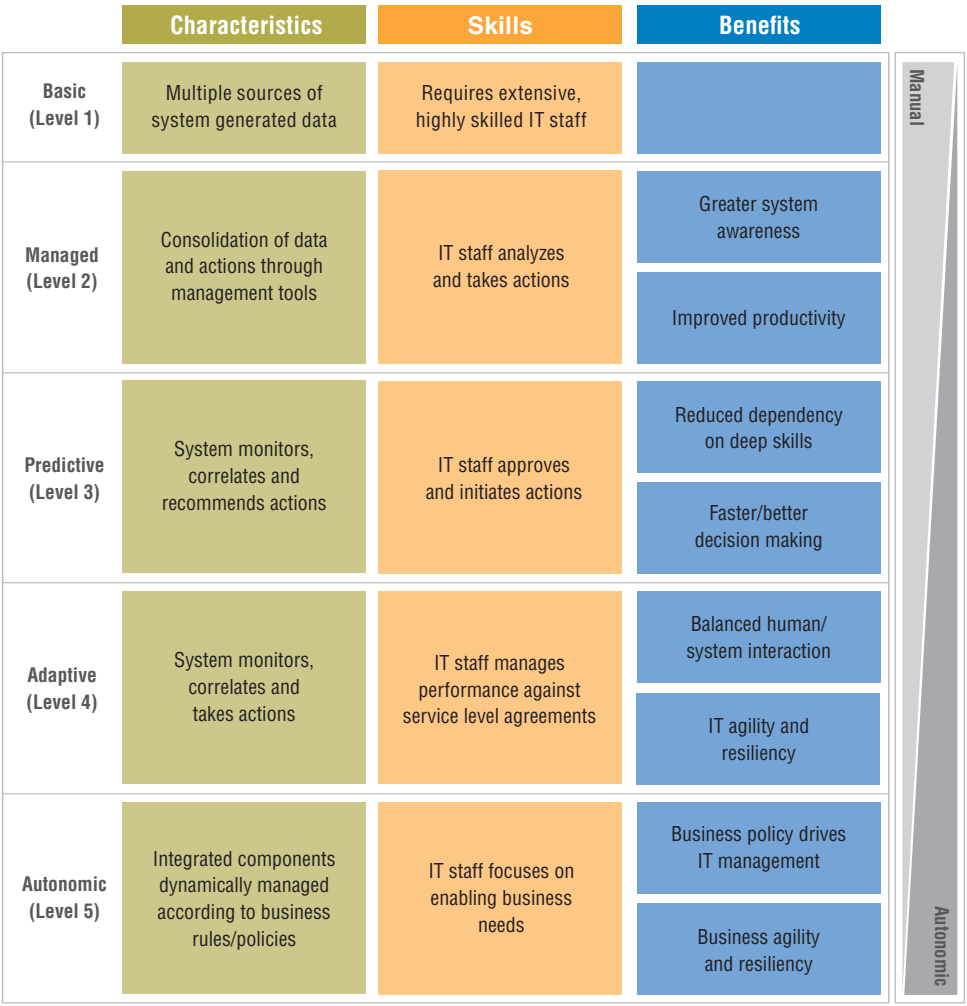

They are:

- *Self-configuring—they can adapt dynamically—with minimal human intervention—to the deployment of new components or changes in the IT environment, providing continuous strength and fostering the productivity of the e-business infrastructure*
- *Self-healing—they can detect improper operations and initiate corrective actions before they occur, without disrupting system applications or business processes, thus allowing greater resilience*
- *Self-optimizing—they can efficiently tailor resource allocation and utilization to meet user needs of the moment and ensure optimal quality of service*
- *Self-protecting—they can detect hostile or intrusive behavior as it occurs, and make themselves less vulnerable to unintentional human error as well as to malicious actions*

**An evolutionary path toward autonomic computing**

Realizing systemwide autonomic environments is an evolutionary process that is enabled by technology. However, it is ultimately implemented by each enterprise through the adoption of these technologies, along with supporting skills and processes. The figure below represents levels of autonomic maturity, beginning at the basic starting point.

Many companies can benefit today by implementing technologies that help them advance through these levels. By the time they reach the final level, their IT systems will be capable of dynamic adaptation based on business policies.



**IBM leads the way**

Experienced in hardware, software, IT services and processes, IBM is well positioned to help companies move to fully implement autonomic computing and realize the benefits of better return on investment, agility and quality of service.

Leveraging technology from IBM's world-renowned Research Division, autonomic computing is now a burgeoning movement driven by the entire company. IBM has not only devoted significant research and development resources and established its intellectual leadership in this arena, it has already been incorporating autonomic capabilities across its product portfolio.

While it may be some time before technology systems become as adaptive as the human autonomic nervous system, many are already exhibiting such qualities and are delivering benefits to the organizations that depend on them. And IBM is leading the effort to evolve hardware and software technologies to become even more flexible, accessible and transparent to users—so you can focus on your business, not technology.

IBM is leading the effort in autonomic computing and is delivering autonomic solutions today.

For more information, please visit: **ibm.com**/autonomic

*dynamically adapt*



IBM United States
Software Group
Route 100
Somers, NY 10589
U.S.A.

Printed in the United States of America
10-02


Printed in the United States on recycled paper
containing 10% recovered post-consumer fiber.

*Tivoli software: Intelligent management software that integrates and automates*

Tivoli software

# Autonomic computing and the Tivoli software storage family

*Helps enable enterprises to operate efficiently, minimizes costs and enhances your company's ability to react to change*

### Introducing autonomic computing

Autonomic computing is the ability of IT infrastructures to adapt to change in a complex environment in accordance with business policies and objectives. This flyer defines the concepts and value of autonomic computing; the autonomic self-managing characteristics and its affect on process, tools and skills within the IT organization; and the evolution of systems and technology in autonomic computing.

Autonomic computing systems have the ability to perform management activities based on situations they observe or sense in the IT environment. Rather than IT professionals initiating management activities, the system observes something about itself and acts accordingly—it becomes self-managing.

Autonomic computing is the self-management of e-business infrastructure, balancing what is managed by the IT professional and what is managed by the system. It is the evolution of e-business.

### Why autonomic computing?

Why is autonomic computing important today? The cost of technology continues to decrease yet overall IT costs do not. With the expense challenges that many companies face, IT managers are looking for ways to improve the return on investment of IT by reducing total cost of ownership, improving quality of service, accelerating time to value and managing IT complexity. e-business is a reality, and outages are proving to be expensive and embarrassing.

Adding to this complexity is the proliferation of heterogeneous vendor and technology environments, which require the components of a given solution

to be integrated and customized into unique customer business processes. The increased need to distribute data, applications and system resources across geographic and business boundaries further contributes to the complexity of the IT infrastructure.

*Can free you from mundane tasks and can help you refocus on defining your company's business priorities*

**What is autonomic computing?**

Autonomic computing is about freeing IT professionals to focus on high-value tasks by making technology work smarter. This means letting computing systems and infrastructure take care of managing themselves. Ultimately, it is writing business policies and goals and letting the infrastructure configure, heal and optimize itself according to those policies while protecting itself from malicious activities.

In an autonomic environment the IT infrastructure and its components are self-configuring, self-healing, self-optimizing and self-protecting. They communicate with each other and with high-level management tools. They regulate themselves and, sometimes, each other in accordance with business priorities. Together they proactively manage the IT environment under the protection of suitable controls while hiding the inherent complexity of these activities from end users.

Collectively these intuitive and collaborative qualities enable enterprises to operate efficiently with fewer human resources, decreasing costs and enhancing a company's ability to react to change. Users are freed from mundane tasks and can focus on defining the business objectives that the autonomic system will use to govern itself.

**How does autonomic computing affect your business?**

No matter how large or how small, IT organizations perform similar sets of tasks to manage their IT environment. Because autonomic computing is about shifting the burden of managing systems from people to technology, it is important to understand these sets of tasks and verify that they are synchronized with autonomic computing.

The business of IT can be described as a collection of practices or processes that organize the work of the IT staff to deliver service. A popular example of this is the IT Information Library (ITIL)—a set of best practices that apply to IT organizations regardless of a company's size or the technology used.

A framework of best-practices disciplines serves as a guide during the shift from you managing your IT infrastructure to the autonomic computing technology handling it. However, it does happen overnight and cannot be solely accomplished through the acquisition of new products. Skills within the organization need to adapt, and processes need to change, creating new benchmarks of success. By coupling best practices with comprehensive enabling technology, organizations can realize substantial economic value.


**What is the value of autonomic computing?**

Autonomic computing helps address market conditions by using technology to manage technology, helping reduce operational costs and improve return on investment. In autonomic computing, IT infrastructure components take on the following characteristics: self-configuring, self-healing, self-optimizing and self-protecting. IT infrastructure components need to expose manageability characteristics to each other and to high-level management tools.

***Self-configuring***

With the ability to dynamically configure itself on the fly, an IT environment can adapt immediately—and with minimal human intervention—to the deployment of new components or changes in the IT environment. Dynamic adaptation helps verify continuous strength and productivity of an e-business infrastructure—often the single-determining factor between business growth and chaos.

***Self-healing***

Self-healing IT environments can detect problematic operations (either proactively through predictions or otherwise) and then initiate corrective action without disrupting system applications. Corrective action could mean that a product

alters its own state or influences changes in other elements of the environment. Day-to-day operations do not falter or fail because of events at the component level. The IT environment as a whole becomes more resilient as changes are made to reduce or help eliminate the business impact of failing components.

***Self-optimizing***

Self-optimization refers to the ability of the IT environment to efficiently maximize resource allocation and utilization to meet end users' needs with minimal human intervention. In the near term self-optimization primarily addresses the complexity of managing system performance. In the long term self-optimizing components may learn from experience and automatically and proactively tune themselves in the context of an overall business objective. Self-optimization verifies optimum Quality of Service for both system users and their customers.

***Self-protecting***

A self-protecting IT environment is one that lets the right people have access to the right data at the right time and can take appropriate actions automatically to make itself less vulnerable to attacks on its runtime infrastructure and business data.

A self-protecting IT environment can detect hostile or intrusive behavior as it occurs and take autonomous actions to make itself less vulnerable to unauthorized access and use, viruses, denial-of-service attacks and general failures. The self-protecting capability allows businesses to consistently enforce security and privacy policies, reduce overall security administration costs and ultimately increase employee productivity and customer satisfaction.

### The results of autonomic computing

The results delivered by implementing autonomic computing solutions with self-managing characteristics include:

***Operational efficiency***

- *As IT infrastructure becomes more autonomic, executing on business policy becomes the focus of IT management. Management of the business and of IT will no longer be separate or possibly conflicting activities.*
- *Self-configuring and self-optimizing technologies drive efficiency in running and deploying new processes and capabilities.*

***Supporting business needs with IT***

- *The actualization of self-configuring systems speeds the deployment of new applications required to support emerging business requirements.*
- *Self-healing capabilities help deliver the 24x7 availability required to keep businesses running.*

***Workforce productivity***

- *Workforce productivity is enhanced when the focus is on management of business processes and policies, without the need to translate these needs into actions that separately manage supporting technology.*
- *Systems that self-manage free up IT resource to move from mundane system management tasks to focus on working with users to solve business problems.*

#### Levels of autonomic computing: an evolution not a revolution

The progression toward an autonomic environment is an evolutionary process that is enabled by technology but is ultimately implemented by each enterprise through the adoption of these technologies and supporting processes. Each facet of autonomic computing—self-configuring, self-healing, self-optimizing and self-protecting—offers unique challenges and opportunities. Customers can benefit from several common values as they progress from level to level, including:

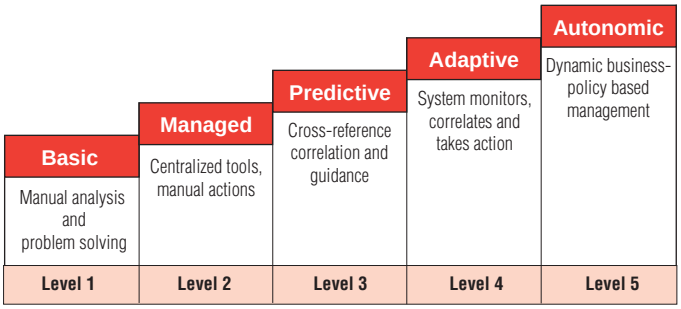
- *Increase critical system availability*
- *Leverage existing technical and staff resources*
- *Improve responsiveness to change*
- *Improve comprehensive performance*
- *Reduce cycle time for deployment of new systems*

*"The Tivoli group at IBM is introducing autonomic computing in a step-by-step, phased manner. This allows customers to leverage their existing technical and staff resources with today's autonomic offerings and enables them to progress to the next level as new offerings appear. This process can help customers achieve a steadily increasing return on their investments."*

—Paul Mason, IDC

Delivering systemwide autonomic environments is an evolutionary process that is enabled by technology, but is ultimately implemented by each enterprise through the adoption of these technologies along with supporting processes.

What's the journey we must take, and what technologies are needed to help us get there? The path to autonomic computing can be thought of in five levels. These levels, defined below, start at basic and continue through managed, predictive, adaptive and finally autonomic.




1. ***Basic level***–*a starting point of IT environments. Each infrastructure element is managed independently by IT professionals who set it up, monitor it and eventually replace it.*
2. ***Managed level***–*systems management technologies can be used to collect information from disparate systems onto fewer consoles, reducing the time it takes for the administrator to collect and synthesize information as the IT environment becomes more complex.*
3. ***Predictive level***–*new technologies are introduced to provide correlation among several infrastructure elements. These elements can begin to recognize patterns, predict configuration and provide advice on what course of action the administrator should take.*
4. ***Adaptive level***–*technologies improve and people become more comfortable with the advice and predictive power of these elements. The IT environment can automatically take the right actions based on the available information and the knowledge of what is happening in the environment.*
5. ***Autonomic level***–*the IT infrastructure operation is governed by business policies and objectives. Users interact with the autonomic technology to monitor the business processes, alter the objectives, or both.*

**Autonomic computing best practices**

The evolution to autonomic computing is not just about improved technologies or tools; changes are needed in skill levels, processes and benchmarks to evolve and move toward autonomic computing. These are represented in the diagram on page 8.

As companies progress through the five levels of autonomic computing, the processes, tools and benchmarks become increasingly sophisticated, and the skills requirement becomes more closely aligned with the business.

The basic level represents the starting point for many IT organizations. If IT organizations are formally measured, they are typically evaluated on the time required to finish major tasks and fix major problems. The IT organization is viewed as a cost center, with variable labor costs preferred over an investment in centrally coordinated systems management tools and processes.

At the managed level IT organizations are measured on the availability of their managed resources, their time to close trouble tickets in their problem management system and their time to complete formally tracked work requests. To improve on these measurements, IT organizations document their processes and continually improve them through manual feedback loops and adoption of best practices. IT organizations gain efficiency through consolidation of management tools to a set of strategic platforms and through a hierarchical problem management triage organization.

In the predictive level IT organizations are measured on the availability and performance of their business systems and their return on investment. To improve, IT organizations measure, manage and analyze transaction performance. The critical nature of the IT organization's role in business success is understood. Predictive tools are used to project future IT performance, and many tools make recommendations to improve future performance.

In the adaptive level IT resources are automatically provisioned and tuned to optimize transaction performance. Business policies, business priorities and service-level agreements guide the autonomic infrastructure behavior. IT organizations are measured on comprehensive business system response times (transaction performance), the degree of efficiency of the IT infrastructure and their ability to adapt to shifting workloads.

| Basic<br>Level 1 | Managed<br>Level 2 | Predictive<br>Level 3 | Adaptive<br>Level 4 | Autonomic<br>Level 5 |
|---|---|---|---|---|
| **Process**<br>Informal, reactive, manual | **Process**<br>Documented, improved over time, leverage of industry best practices, manual process to review IT performance | **Process**<br>Proactive, shorter approval cycle | **Process**<br>Automation of many resource management best practices and transaction management best practices, driven by service-level agreements | **Process**<br>IT service management and IT resource management best practices are automated |
| **Tools**<br>Local, platform and product-specific | **Tools**<br>Consolidated resource management consoles with correlation of events, problem management system, automated software install, intrusion detection, load balancing | **Tools**<br>Role-based consoles with analysis and recommendations; product configuration advisors; realtime view of current and future IT performance; automation of some repetitive tasks; common knowledge base of inventory and dependency management | **Tools**<br>Policy-management tools drive dynamic change based on resource-specific policies | **Tools**<br>Costing/financial analysis tools, business and IT modeling tools, tradeoff analysis; automation of some e-business management roles |
| **Skills**<br>Platform-specific, geographically dispersed with technology | **Skills**<br>Multiple platform skills, multiple management tool skills | **Skills**<br>Cross-platform business system knowledge, IT workload management skills, some business-process knowledge | **Skills**<br>Service objectives and delivery per resource, analysis of impact on business objectives | **Skills**<br>e-business cost and benefit analysis, performance modeling, advanced use of financial tools for IT context |
| **Benchmarks**<br>Time to fix problems and finish tasks | **Benchmarks**<br>System availability, time to close trouble tickets and work requests | **Benchmarks**<br>Business system availability, service-level agreement attainment, customer satisfaction | **Benchmarks**<br>Business system response time, service-level agreement attainment, customer satisfaction, IT contribution to business success | **Benchmarks**<br>Business success, competitiveness of service-level agreement metrics, business responsiveness |

*"By focusing on staged delivery and what is actually available today and tomorrow, Tivoli software can help its customers work toward the autonomic computing vision."*

—James Governor, Illuminata

In the autonomic level IT organizations are measured on their ability to make the business successful. To improve business measurements they understand the financial metrics associated with e-business activities and supporting IT activities. Advanced modeling techniques are used to optimize e-business performance and quickly deploy newly optimized e-business solutions.

What should you do next in assessing your self-managing environment?
Now that we described what autonomic computing is and how you can benefit from it, your next step is to assess your own environment from a self-management perspective. IBM can help you do just that.

#### Leveraging open standards

Open standards are essential for managing resources and processes across the system hierarchy layers. IBM leverages its leadership in open standards as the foundation for its autonomic system management solution. These standards currently include:

- *Java™ Management Extensions*
- *Web service-level agreements*
- *Storage Networking Industry Association*
- *Open-grid systems architecture*
- *Web Services standards*
- *Telecom management*
- *Distributed Management Taskforce*
- *Common Information Model*
- *Internet Engineering Taskforce (Policy, Simple Network Management Protocol)*
- *Organization for the Advancement of Structured Information Standards (OASIS)*
- *Microsoft® Windows® management instrumentation*

#### Tivoli Storage products can make autonomic computing a reality today

Tivoli® software from IBM provides a head start toward reaching the ultimate objective of a fully autonomic system and continues to participate in developing the ultimate solution. Many Tivoli management disciplines provide some level of automation that can help achieve the potential benefits of autonomic computing. Tivoli products support and incorporate the autonomic values of

self-configuring, self-healing, self-optimizing and self-protecting. Following are a few examples of individual product capabilities.

***Self-healing capabilities***

Tivoli Storage products survey and analyze storage resources networkwide, providing key information on storage assets, capacity, utilization and availability. The scope and criteria of storage issues or problems can be defined to suit different situations. For example, alerts are issued to the storage administrator if free space falls below a user-defined value, an unrecoverable file is discovered, storage usage exceeds a user-defined value, a file that has not been accessed in a user-defined length of time has been discovered or a storage resource is added or removed from the network. Tivoli Storage products initiate self-healing and policy-based corrective actions, including triggering automatic backups to correct potential threats and verify that day-to-day operations run smoothly.

*Business benefits:* Self-healing capabilities provide a resilient storage environment as changes are made to reduce or help eliminate the business impact of failing components.

*"The self-configuring capabilities of Tivoli Storage Manager help us optimize data protection across all our applications."*

HwangYong Seung, Chief Information Officer, Korean Air

***Self-configuring capabilities***

Tivoli Storage products provide self-configuring capabilities to perform such tasks as automatically identifying and loading the appropriate drivers for storage devices connected to servers. The server software also looks across the network to identify other storage management servers and initiate communication links with them to allow a server to assume control. The client recognizing the file system that the host computer is using and identifying the network domain it resides on to make the appropriate configuration changes also achieves self-configuration.

*Business benefits:* Self-configuring allows the storage environment to easily adapt—and with minimal human intervention—to the deployment of new components or changes.

***Self-protecting capabilities***

The ultimate objective of Tivoli Storage products is to protect the enterprise's most valuable asset—the data. Tivoli Storage products protect data from tampering or nonauthorized use by both data encryption and a Kerberos-like

authentication and authorization scheme. If a significant change in a database has occurred since the last data backup process, it sends alerts to the administrator to warn of potential trouble. Data overruns are detected and prevented when a certain percent of a data recovery log is used. It automatically triggers a policy-based data backup to free storage space. A copy of storage pools is also generated to offline and offsite storage to help protect against local disk failures.

*Business benefits:* The self-protecting capability allows businesses to reduce overall storage administration costs and help increase employee productivity and customer satisfaction.

***Self-optimizing capabilities***

For configurations that include both storage area network and local area network (LAN) connections, Tivoli Storage products will automatically fail-over and revert to LAN-free data movement. Self-optimizing features such as progressive backup can help you get the most from network bandwidth, media and storage staff—increasing the efficiency of the enterprise. Instead of transferring an entire disk volume, Tivoli Storage products are built to recognize and transfer only those data files that have changed since their last data backup. Subfile differential backup further enhances this process and provides efficiency by transferring only those bytes or blocks of data (which make up each file) that have changed since the last data backup. The process can be self-optimized by using compression and appropriately balancing data transfers across multiple network connections.

*Business benefits:* Self-optimization helps provide optimum Quality of Service for storage users.

**The Tivoli Storage Manager family of products**

***IBM Tivoli Storage Manager***

Tivoli Storage Manager provides a powerful suite of automated tools for enterprisewide data protection. With its autonomic functionality, Tivoli Storage Manager can help organizations protect information and maximize the availability of key applications. By offering comprehensive, policy-based data protection and data restore capabilities, it can help minimize the probability of data loss and help maximize the productivity of your business resources.

## *Why Tivoli software products for Autonomic Computing?*

- *Because Tivoli software delivers autonomic capability today*
- *Because Tivoli software frees customers to focus on their business instead of their infrastructure*
- *Because Tivoli software combines the agility of a startup with the experience, quality and resources of IBM*
- *Because Tivoli software provides step-by-step roadmaps*
- *Because Tivoli software represents evolution not revolution*
- *Because the Tivoli software team is ready to assist (IBM and IBM Business Partners)*

***IBM Tivoli Storage Resource Manager***

Tivoli Storage Resource Manager is a premier Java and Web-based solution that can help identify, evaluate, control and predict enterprise storage management assets. Because it is policy-based and has autonomic self-healing capabilities, it can detect potential problems and automatically make adjustments based on the policies and actions established by the enterprise.

## Summary

Companies want and need to reduce their IT costs, simplify management of their IT resources, realize a fast return on their IT investment and provide high levels of availability, performance, security and asset utilization. Autonomic computing addresses these issues, and it is not just about new technology. Autonomic computing is about freeing IT professionals to focus on higher-value tasks by making technology work smarter. It can be accomplished through a combination of process changes, skills evolution, new technologies, architecture and open industry standards.

With autonomic computing reducing the demand for the specialized skills currently required to manage IT initiatives, projects are more likely to be implemented on time and on budget. The incorporation of autonomic technologies can lead to systems that deliver the expected quality of service, helping eliminate many configuration problems introduced in today's environments due to human error, reducing the man outages caused by malfunctions that can now be self-corrected and delivering maximum results and performance by continually optimizing the use of resources.

*For more information on the IBM autonomic computing strategy and integrated solutions from IBM, contact your IBM sales representative or visit* **ibm.com/tivoli**



IBM Corporation
Software Group
Route 100
Somers, NY 10589
U.S.A.

Printed in the United States of America
10-02


The Tivoli home page on the Internet can be found at **ibm.com/tivoli**

The IBM home page on the Internet can be found at **ibm.com**

♻ Printed in the United States on recycled paper containing 10% recovered post-consumer fiber.

G325-6855-00

# Autonomic personal computing

by D. F. Bantz
C. Bisdikian
D. Challener
J. P. Karidis
S. Mastrianni
A. Mohindra
D. G. Shea
M. Vanover

Autonomic personal computing is personal computing on autonomic computing platforms. Its goals combine those of personal computing with those of autonomic computing. The challenge of personal autonomic computing is to simplify and enhance the end-user experience, delighting the user by anticipating his or her needs in the face of a complex, dynamic, and uncertain environment. In this paper we identify the key technologies that enable autonomic behavior as distinguished from fault-tolerant behavior. We give some examples of current autonomic behavior and some general considerations for an architecture that supports autonomic personal computing. We identify its challenges to standards and technology developers and conclude with some guidance for future work.

Autonomic personal computing is defined here as personal computing on autonomic computing systems. It shares the goals of personal computing—responsiveness, ease of use, and flexibility—with those of autonomic computing—simplicity of use, availability, and security. In most cases these goals are complementary. For example, autonomic computing enhances ease of use because it eliminates or simplifies some user responsibilities. But personal computing implies flexibility of location and of the hardware and software configuration, and this complicates the job of achieving autonomic behavior. It is easier to configure, heal, optimize, and protect a system in an environment that does not change. If we can achieve autonomic behavior while still meeting the unique needs of personal computing, millions of users will benefit worldwide.

The intention of this paper, then, is to identify the unique demands and opportunities of autonomic computing with personal devices. Our ground rules are that we seek to achieve autonomic behavior of a personal computing *system*—personal computers (PCs) and their peers, networks, and servers—not just the PC alone. We also limit our focus to application *platforms*, not to applications themselves. This distinction is somewhat equivocal and quantitative, however, because yesterday's applications are often tomorrow's platforms.

In what follows, we first look deeper into the meaning of the autonomic attributes of personal computing, which are different from fault-tolerant attributes. We then categorize technologies as they relate to achieving autonomic behavior in different variations: within the PC, in PC communities, and in more general systems that include servers. We give some examples of the state of the art and identify missing or incomplete capabilities. We describe some general considerations for an architecture that supports autonomic personal computing, identify some issues, and suggest a direction for future research and development.

## Autonomic personal computing

A computing system is autonomic if it possesses at least one of four attributes: self-configuring, self-healing, self-optimizing, and self-protecting. Autonomic personal computing exhibits and constrains these attributes in unique ways.

**Self-configuring.** A system is self-configuring to the extent that it automates the installation and setup of its own software in a manner responsive to the needs of the platform, the user, the peer group, and the enterprise. Personal computing often involves user-initiated configuration change, and a self-configuring system understands the implications of these changes and accommodates them automatically.

**Self-healing.** A system is self-healing to the extent that it monitors its own platform, detects errors or situations that may later manifest themselves as errors, and automatically initiates remediation. Fault tolerance[1] is one aspect of self-healing behavior, although the cost constraints of personal computing often preclude the redundancy required by many fault-tolerant solutions.

**Self-optimizing.** A system is self-optimizing to the extent that it automatically optimizes its use of its own resources. This optimization must be done with respect to criteria relevant to the needs of a specific user, his or her peer group, and the enterprise. Resource management[2] is one aspect of self-optimizing behavior.

**Self-protecting.** A system is self-protecting to the extent that it automatically configures and tunes itself to achieve security, privacy, function, and data protection goals. This behavior is of very high value to personal computing, which is exposed to insecure networks, an insecure physical environment, frequent hardware and software configuration changes, and often inadequately trained end users who may be operating under conditions of high stress. Security[3] is one aspect of self-protecting behavior.

## Examples of current autonomic personal computing behavior

Autonomic behavior is not new; computing systems have incorporated various forms of autonomic behavior for many years. Autonomic function creates autonomic behavior.

First, we introduce a categorization of autonomic function in terms of where it is implemented, and then we discuss several examples that exhibit autonomic behavior in current practice.

Autonomic function can be implemented locally, drawing on locally maintained measurements and knowledge. It can be implemented among members of a peer group, sharing measurements and knowledge particular to that group. It can also be implemented using globally available network-resident resources, in which case, measurements and knowledge are maintained for all clients. In the most general case, autonomic functions are implemented in all three ways, with different functions having a preferred implementation, resulting in the following categories:

- Local autonomic function—Locally, autonomic decisions can be made using knowledge that the personal computer stores or can obtain by itself, for example, from its Common Information Model (CIM)[4] database. Local functions include automatic auditing of software configurations, local backup, surveys of the connectivity environment, and power management.
- Peer group autonomic function—Peer group functions require the cooperation of a local community. They include spontaneous grid computing services and knowledge sharing.
- Network-based autonomic function—Network-based functions enhance and extend the core autonomy of the PC. Examples include software updating, backup and restore, virus updates, and mobility support services.

**Local autonomic function.** Microsoft Corporation has included some local autonomic features in the Windows** XP[5] operating system, and many other autonomic features are provided by third-party utilities. Although an exhaustive review is beyond the scope of this paper, in the following subsections we describe some examples of local autonomic behavior that can be found in current practice.

***Installation, configuration, and maintenance.*** The life cycle of a personal computer begins when it is delivered, set up, and personalized to the needs of its new user. Part of this personalization involves creating a replicable software configuration appropriate to the needs of a group of users (imaging), and part involves selectively moving the user's data and preferences from a previous platform to the new one (migration).

Technology now exists for simplifying the imaging process. In IBM's ImageUltra, a single hardware-independent super-image is created and distributed to the PC. The super-image adapts to the platform and is customized to user needs based on a key. This key, which may be specific to a particular user or common to a group, is either distributed electronically or entered manually by the user or an administrator. A utility program transforms the super-image into the final one.

Once the system has the correct image, the user's specific information, settings, and application files need to be transferred. Windows XP includes a utility for accomplishing this transfer, with a good deal of human intervention. IBM has a comparable solution (the System Migration Agent) for Microsoft operating systems prior to Windows XP. The core logic for this product is still central to IBM's large enterprise solution for automating the migration of user content for large numbers of systems. The Ghost** product from Symantec Corporation also implements image capture, redeployment, and migration.

Change management deals with updates to system and application software after the initial installation. The Microsoft XP Automatic Update mechanism works well for updates that can be applied without special considerations, but some enterprises require tight policy-driven control over their configurations. The IBM Update Connector supports either user-initiated or centrally administrated updates. The CNET Networks, Inc. CatchUp is a personal Web-based service that automatically analyzes the software configuration of a system and identifies needed updates.

Break/fix primarily concerns situations in which the system was once in the correct state and needs to return to that state or one close to it. The Microsoft Windows Installer saves a valid state for the core of the operating system and for selected applications. The saved state is accessible to the user, and thus vulnerable to user error or to a security breach. IBM's solution, Rapid Restore PC, saves the complete state on a hidden partition of the hard drive of a system. Both of these solutions have only limited autonomic behavior because they require informed human intervention.

***Communications.*** Some automation of communications tasks has become possible because of networking support services based on open standards. Using the Dynamic Host Configuration Protocol (DHCP) and domain name system (DNS) services, client devices self-configure in a network based on Transmission Control Protocol/Internet Protocol (TCP/IP).

Unfortunately, other parameters are not generally available from these services. Mail server addresses, Web proxy addresses and types, security settings such as for virtual personal networks, and wireless access point settings all require manual intervention to configure today.

Further complicating the situation is the proliferation of communication technologies and standards, and the inclusion of support for multiple networks in today's mobile personal computers. Windows XP is aware of the presence or absence of each network interface, and whether or not it is connected to "active" media, that is, whether any communication is possible with the device that is one hop away. This awareness enables some autonomic behavior, for example, automatic "failover" to another network in the case of a cable fault.

***Self-optimization.*** Windows XP Professional modifies the user interface based on the way in which the system is used. Instead of an alphabetically sorted list of programs, the user is presented with a list of the most recently used programs. Shortcuts to these programs are placed in a reserved area that allows the programs to be subsequently launched with a single click. XP also attempts to keep the desktop clean and uncluttered by consolidating the items that appear on the Windows taskbar. If multiple files are opened by the same application, the files are consolidated on the taskbar.

As another example of self-optimization, Norton Utilities** senses the current level of disk file fragmentation and alerts the user if performance might be degraded. It automatically reorganizes the physical placement of data on the disk to improve file access.

***Self-protection.*** The user's data are a key asset and potentially vulnerable to both failures and attacks. The first line of defense is to back up those data, and many systems exist to implement backup and restore. Backup solutions in current use direct their technology toward reducing their resource usage (storage and bandwidth) as well as toward automating the initiation of the backup itself. Backup can be scheduled periodically or can be initiated proactively in response to a hardware-initiated event, such as detection of incipient disk failure.

In the communication and storage of data, encryption is another example of a proactive protection mechanism. Data are encrypted at the time of creation, decrypted upon use, and then re-encrypted when done. Windows XP includes an encryption capability that allows selected directories and subdirectories to be encrypted. Key management remains an issue. Some IBM personal computers contain a hardware security solution, called the Embedded Security Subsystem, that generates and maintains unique encryption keys. The randomness and hardware-enforced protection of these keys supports a broad class of protection mechanisms, including digital signature, signature verification, bulk encryption, secure e-mail, and password protection.

**Peer group autonomic function.** Although local autonomic function depends on (hopefully) authoritative local data, and network-based autonomic function depends on data maintained by a responsible authority, peer group autonomic function depends on information solicited from other personal systems, which may or may not be accurate, timely, or relevant.

Current behavior among members of a peer group (e.g., personal computers on the same network segment) is usually confined to resource sharing—files, printers, and other peripheral devices. Here, the community of interest, the tie that binds the peer group together, is defined by the need to share information and to reduce costs by sharing expensive resources that are otherwise underutilized. Several technologies are available that allow users to discover and use resources available in the peer groups. The Microsoft Windows Browse Master allows users to discover shared folders, printers, scanners, and other resources that other Windows-based personal computers export. Systems such as Gnutella[6] have focused primarily on enabling anonymous file-sharing among peer-group clients.

Autonomic behavior can be created on the basis of knowledge that is discovered or acquired from a peer group too. For example, some connectivity parameters (e.g., proxy server addresses) are hard to discover, but because they may be known in the peer group, they may be propagated to needy members. The community of interest for knowledge-sharing is quite specific to the type of knowledge desired. The peer group for the discovery of proxy server addresses would be, for example, all members that access the Internet.

The YouServ[7] tool is a simple but potentially useful first step toward peer knowledge-sharing. YouServ uses existing Web technologies to achieve a very easy-to-use system with a very low-cost implementation. Most recently, the research community has experimented with "grid computing,"[8] a new field that focuses on large-scale resource sharing among virtual organizations. Grid technologies have focused on building protocols, services, and tools to enable virtual organizations. We believe that for peer-group autonomic function to become a reality, technologies similar to those used in the grid need to be developed to enable virtual communities—communities of autonomic clients that collaborate to solve problems related to autonomic computing.

**Network-based autonomic function.** Network-based services can help PCs achieve autonomic behavior. These services can provide information to local autonomic function, provide function that complements local function, or, in some cases, can even replace local function. IBM's Access Support is an example of a service that supplies support information (e.g., BIOS [Basic Input/Output System] and device driver updates) given information about the hardware and software configuration of the PC.

Remote backup is an example of a service that complements local data protection function by providing additional storage in a physically remote location. The current generation of network-based services, specifically Internet-based services, often requires direct end-user interaction to use the service. Web services,[9] an open standard based on the Extensible Markup Language (XML) for computer-to-computer services, obviate the necessity of human involvement, permitting the use of network-based services in a more autonomic manner.

Although some elements of a sufficient platform exist for network-based autonomic function, they have yet to be put together into a coherent partner for local or peer group autonomic personal computing. In the next two sections we explore a possible architecture that links these three types of autonomic function, and we explore the challenges that each type must meet in order to achieve seamless autonomic behavior.

## An architecture for autonomic personal computing systems

The architecture of an autonomic system, including that of a personal computing system, begins with the

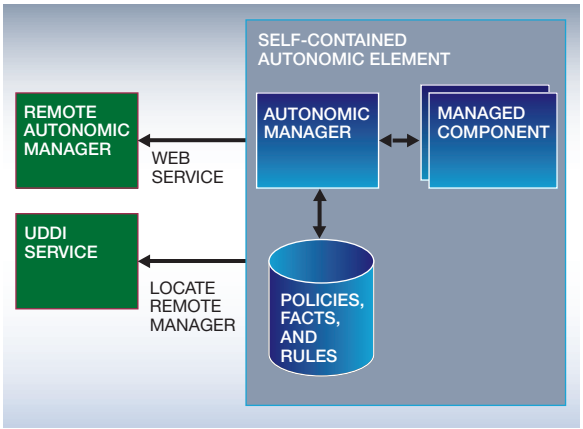
general architecture for autonomic systems.[10] The building block of autonomic systems is depicted in Figure 1.

Figure 1 shows the architecture of an autonomic element (AE). Each AE consists of an autonomic manager (AM) and a set of managed components. Each managed component is responsible for communicating its events and other measurements to the local AM. In turn, based on the input received from each managed component, the AM makes decisions taking into account its policy, facts, and rules (stored locally in a database) and communicates the directives and hints to the managed component.

The figure makes a distinction between self-contained autonomic behavior (within the large gray box on the right) and autonomic management involving explicit communications with a remote manager. The interfaces between an AM and its managed component are an important part of the architecture. For Windows-based personal computing systems, one extensive set of these interfaces is represented by the Windows Management Instrumentation,[11] or WMI. The interface between a remote AM and an AE is not currently standardized. This interface must be discoverable and dynamically bound so as to support self-configuration of autonomic systems; it must also be secure and private. The figure shows a remote autonomic manager implementing a Web service, located via the Universal Description, Discovery, and Integration (UDDI) service registry. We see Web services as a foundation technology because they provide standard ways to locate, communicate (via XML), compose, and interact with network-based services. But because personal systems are often mobile and occasionally disconnected, the interface must support a disconnected (off-line) mode of use as well.

Figure 2 shows the architecture of an autonomic system consisting of autonomic elements connected to one another at local, peer, and network levels. Resources are shown as boxes, AMs as diamond shapes, peer groups as dashed ellipses, and physical resources—servers and clients—as circles. Arrows represent the control exerted by AMs (e.g., S controls W). At the local level, there is a single AM (e.g., A) that is capable of independent decision-making. At the peer level, each AM interacts and shares knowledge and information with its peers and may act cooperatively, as though a virtual autonomic manager (e.g., W) is present.

Figure 1 Architecture of an autonomic element

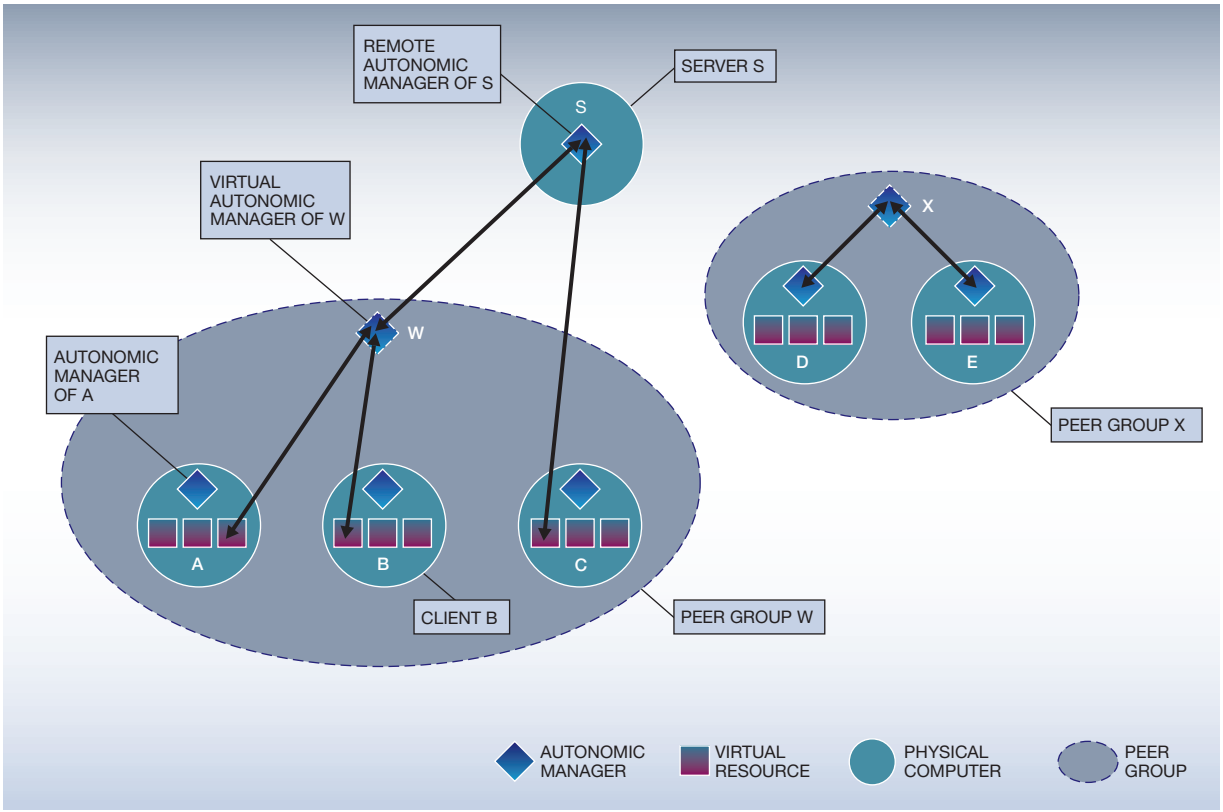



The notion of a virtual autonomic manager is to have the participants in a peer group achieve some level of autonomic behavior as if directed by a local or remote instance of an autonomic manager, even though none is present. The behavior is a consequence of peer knowledge-sharing, obtaining local consensus, and acting locally on that knowledge. The elements of a peer architecture, discovery, query, and binding, support this process.

One of the central issues of autonomic personal computing is how autonomic managers at some higher level in the system exert control over lower-level elements—what autonomic elements are controlled, how much (and how rapidly) control is exerted, and what kind of control is performed. We identify two styles of control: delegation and guidance. In delegation, a local autonomic manager passes control of some of the resources it manages to a superior. By default, the control of all local resources is delegated to the local AM. In guidance, a local autonomic manager receives information (e.g., policies) from its superior and implements them with respect to its own resources. Only one AM is ever in direct control of a resource.[12]

Today's personal computers manage all of their resources with a single manager—their operating system. It is advantageous to support a model in which control over some resources can be delegated to another manager, and this is possible with client virtualization. If we think of the boxes in Figure 2 as virtual machines, we see that two of the virtual machines of client A are under complete local control,

Figure 2 An example of two hierarchies of autonomic control




whereas the third is under peer group control. Client C has two locally controlled virtual machines and one managed centrally, perhaps by the information technology (IT) service and support department of the enterprise.

In the case of occasional disconnection from a remote manager, a simple static strategy has the remote manager provide policy guidance but not real-time control. An example comes from security management. A remote, global security manager can be aware of inter- and intra-enterprise issues (e.g., a new style of attack) that a disconnected client cannot discover until the critical moment of first reconnection. At that moment, a race exists between the update from the remote manager and the attack, suggesting that the client take special precautions to securely obtain the latest security guidance before resuming its normal local security policy. Such strategies support the traditional personal computing style of considerable local autonomy.

We now discuss two issues that we consider fundamental to the success of this architecture: security and privacy on the one hand, and stability on the other.

**Security and privacy.** Security and privacy are critical system attributes. In order for a machine to request confidential information from another machine, it needs to be able to securely identify itself. There are several ways in which this can be done, but the easiest (and probably most secure) is to use a public/private key pair. The Trusted Computing Platform Alliance (TCPA)[13] provides an ideal way of securing private keys for machine-level identification. Each system can then identify other machines either by using registered keys kept in a database, such as in the IBM Tivoli Policy director, or by using certificates.

Once identification is completed, a secure connection needs to be established between the comput-

ers. Internet Protocol Security (IPSec)[14] is the standard way of making this connection, but client/server authenticated Secure Socket Layer (SSL)[15] works too, as in TLS.[16] This provides for authenticated requests, confidential transmission of the data, and integrity of the data returned. Executables should be both signed—so that they may be verified as coming from a trusted source—and run in a sandbox.

A higher-level autonomic manager should not push information to a subject manager without it being requested. When a system is in "push" mode, the recipient needs to be able to throttle the behavior of the pushing system to avoid denial of service attacks.

In order for a machine to publish information, it needs to be able to determine whether information is confidential. Nonconfidential information must be explicitly identified; everything else is assumed to be confidential.

**Stability.** The goal of an autonomic personal system is to exhibit both autonomic and usable behavior to its end user. When autonomic components are put together in a system, it is not a given that the system as a whole exhibits stable behavior. For example, a personal computer may make a local evaluation that a shared communications link has high bandwidth, based on characteristics of the media, and may initiate a bandwidth-consuming activity. If other personal computers make the same decision, the link can quickly saturate and become unusable to all of them. A more conservative strategy, say, to defer bandwidth-consuming activities until a record of link performance has been established and only then to attempt more aggressive exploitation of the resource, might yield better overall behavior. In general, we imagine that constructive autonomic behavior will be constrained by context and policy. What context and which policies (and how to represent and maintain them) are important topics for future research.

## Issues and directions in open autonomic personal computing

In this section, we describe several challenges that available technologies do not address. Challenges exist in the areas of security, connectivity, storage, peer group collaboration, network-based services, and the user interface, and autonomic managers, virtualization, and standards. Autonomic frameworks can be brought to bear to meet them.

**Security.** Security solutions are notorious for being difficult to use.[17] Public Key Infrastructure (PKI) is theoretically a very good security design, but it is difficult to implement. Secure keys must be generated for all users and distributed to them. Certificates must be generated for all the keys, and a database of revoked certificates kept. Passwords are insecure and expensive; typically, over 50 percent of the calls to a help desk are requests to reset a password. Even biometric identification requires fingerprint templates, which must be managed. Since this management is likely to be frequent, it represents a significant security exposure.

We believe that the current lack of ease-of-use of security schemes remains a significant retardant to their widespread acceptance. There are many opportunities to apply autonomic computing technologies to security problems, among them automatic updating of security settings, secure recovery from software failures, discovery and remediation of security exposures, and the ability to manage keys securely without human intervention.

Wireless technology presents its own set of challenges, as in the widely reported problems with Wired Equivalent Privacy (WEP).[18] But vulnerable security solutions are not the only source of exposures. Anyone can go to the local computer store and purchase a wireless access point, set it up in an organization, and thereby permit insecure access to the network of the organization. In the wireless arena, convenient auditing means are required, perhaps implemented as portable computers with wireless connectivity and additional instrumentation software, to detect and report these "rogue" access points and improper configuration of those installed by the organization. Wireless technology represents a new security exposure, and autonomic managers should manage its security as well.

**Connectivity.** There are so many alternatives to connect to personal, local, and wide area networks that this choice creates connectivity overload for users. Mobility also adds complexity; each location has its own connectivity attributes. The active communication links of a device should be readjusted to the most appropriate ones each time the status of any of them changes, where a measure of appropriateness would be dependent on quality of service, cost, security, availability, location, and other policy elements. For example,

- If a device is relocated, it should automatically update its connectivity parameters to incorporate past knowledge of the new location; if it has never been in this location before, or does not know where it is, it should try educated guesses (and ultimately, trial and error) to achieve connectivity.
- Connections should be chosen according to policies. For example, a link may be chosen if its cost per bit per second is less than any other, subject to a minimum acceptable bandwidth. If the PC is operating on batteries and remaining power is low, the link with the lowest power requirement may be chosen.

There is a strong interaction between autonomic behavior of connectivity and security. When switching links, active sessions may have to be migrated, exposing the session to penetration. As we share knowledge with peers, we must address the issue of what can be shared with which peers. Personal computer file systems do provide access control per folder, but not an associated security classification. Although the technology exists to secure sharing, the policies and information to control that sharing are generally lacking.

**Storage.** Autonomic storage will start with automation of the storage management that users perform today. Data are often stored in multiple locations and in multiple versions, and it is too easy to lose track of where the data are located. As information is migrated and copied, significant privacy and security requirements must be met. This means that manual work, if not automated, will most likely retard the needed flow of such information.

The challenge in storage is to abstract and manage both the physical location of data and the privacy and security requirements of the data. This abstraction should allow applications developed without autonomic storage in mind to function normally, but perhaps not as optimally as an application designed for autonomic storage. It should also provide the abstraction required for mining collaborative information. Initially, there could be some manual administration of autonomic storage through a simplified interface for setting the physical location (e.g., "add another network attached storage device") and for setting privacy and security parameters. As autonomic storage develops, most or all of this management function should become automatic, guided by higher levels of management that implement broader business-based policy.

**Peer group collaboration.** The reward for peer group collaboration is access to more, and more current, data that may not be available through more formal publishing means. Peer computing is a special case of distributed computing[19] with several challenges. The first is how to form a peer group. Since autonomic function often demands implementations that do not involve human guidance, the peer group must be formed automatically. A given PC should not be constrained to be a member of only one peer group. The peer group for sharing knowledge about how to connect to the Internet from a particular place is almost certainly local to that place, whereas the peer group for sharing resources to support some grid computation need not be. It may be possible to dynamically form a peer group from a larger one (specialization) through a solicitation process in which responses to successively more specialized queries would qualify members, although the indiscriminate broadcast of the first solicitation bodes ill for systems of large scale. Limited persistence of peer groups can limit the need to solicit broadly.

A second challenge is identifying the specific collaboration type for the peer group. How can sharing be limited to just that type? How can solicitations be made both general, so that only a limited set of responses need be designed, and sharp, so that only relevant responses are generated? How are responders to generate helpful responses without compromising their own privacy?

A third challenge is determining the degree of trust that any member puts in the information obtained from any other member in the group. How can a member profit from information so obtained without complete trust in it? Consensus algorithms[19] can derive plausible results, but the credibility of the responders should be taken into account. Credibility can be established through a history of acquiring useful and accurate knowledge from a member, but if no assessment of credibility is available, the uses to which obtained information can be put must be limited.

**Network-based services.** The opportunity for network-based services to complement and extend services implemented locally and in the peer group is too extensive to survey here, but several examples should indicate this potential.

To complement autonomic connectivity (see the earlier subsection on communications), network-based services can supply additional information about re-

sources available in the current location, such as IT resources (printers, hubs, and enhanced displays). Perhaps the key network-based service is the service directory, from which a menu of services and how to access them can be obtained.

An analyst for International Data Corporation asserts that "The market for providing IT services is undergoing a radical change—from provisioning these services as customized offerings, generally at a customer's site, to providing these same services from remote locations as a set of computing utility offerings . . ."[20] An important step in the evolution of autonomic personal computing is the development and deployment of a remote client management utility.[21] The autonomic personal computer facilitates this utility by reducing the need for remote management to exception cases.

The utility itself consists of a secure, scalable autonomic computing infrastructure that can maintain service levels without incurring the costs of excess capacity. This infrastructure will enable capacity on demand and managed services that monitor and resolve issues before they become problems. Proactive management for problem determination, diagnosis, and resolution helps not only to reduce human involvement but also catches incipient failures early, perhaps before they precipitate cascading failures, thus reducing the overall downtime.

As client management utilities evolve, we see a continuing rebalancing of the provisioning of capabilities from the backend infrastructure, from peers, and from those resident on the client. For users to embrace a client management utility, they must be provided with end-to-end security that secures their enterprise data, coupled with sufficiently powerful remote management capabilities to enable ongoing program determination, diagnosis, and resolution without the intervention of the end users.

**User interface.** Many autonomic computing functions have no end-user-visible behavior.[22] They implement "computing that just works." For some users, this behavior is ideal. But for the experienced or the curious, who want to take direct control of their system parameters occasionally, it is important that they be kept up-to-date on autonomic actions that affect these parameters. How can the user be informed of just these actions? There is also the question of how the value of autonomic computing technology is perceived when the activities that deliver that value are hidden.

We believe that it is likely that autonomic personal computing will go through several stages of evolution, differentiated in part by the degree to which the end user is aware of and participates in management actions. As autonomic behavior becomes more effective, it will be trusted more, and the need for an end user to take direct control will lessen. During this time, users will likely be required to select or confirm actions that may be suggested by an autonomic manager.

**Directions toward an autonomic framework.** To address the issues raised in the previous subsections, we are defining an autonomic framework that brings together disparate computing elements. The key elements of the autonomic framework are the autonomic manager and the elements to be managed. The goal of the framework is to specify the interfaces and protocols for elements to exchange information and data to enable collective autonomic behavior. To achieve this goal, we need to rethink the structure of the system and the application software (and the tools that help build them) so as to identify and expose relevant and accurate indications of the state of each element, and provide standard interfaces to affect element state with minimal side effects. Each element will need an element-specific autonomic manager to monitor and control the element. This coupling of element and specific manager represents the lowest level of autonomic behavior. Elements may be isolated in virtual machines to limit undesirable interactions between them. Element-specific autonomic managers will report to a system-wide autonomic manager in a standard way. The system-wide manager is responsible for achieving end-user goals in accordance with established policy.

***Autonomic managers.*** The elements under control of an autonomic manager must be observable and controllable. Current personal computing systems maintain a wealth of data about themselves in repositories and logs. Some of these data are redundant and confusing, and some are not even accurate. Thus the information relevant to decision-making is a challenge to obtain from those data. Similarly, many points of control exist, but their relationship to the desired behavior of the system is unclear. The job of the autonomic manager would be simplified if its input data were more indicative of root causes, and if the actions that the platform supports had a more direct effect on those causes.

Another issue in the design of autonomic managers is the representation and maintenance of knowledge:[23] relations between input data and root causes, relations between output actions and effects, and policies constraining acceptable and desirable actions. This knowledge is often represented in rules, yet rules-based systems are notoriously hard to maintain because of the interactions between rules.

***Virtualization.*** Virtual machines[24] create an efficient emulation of the environment in which operating systems run, at the application protection level. This emulation permits an existing, unmodified operating system to run as an application. A virtual machine monitor allocates resources to virtual machines. Virtual machines encapsulate their contents, so that programs running within them cannot change any state outside of them.

Virtualization enhances isolation and containment, enhancing security and reducing the domain of an autonomic manager to the contents of a virtual machine. Virtualization provides finer-grained resource management and a mechanism for the capture of a complete state, so complex multiapplication execution environments can be frozen, restarted, undone, and migrated or cloned elsewhere. Virtualization provides an effective way for legacy software systems to coexist with current operating environments.

***Standards.*** For autonomic personal computing to succeed, open standards must address the requirements of autonomic computing. Current standards activities on Resource Description Framework (RDF), Semantic Web, Simple Object Access Protocol (SOAP), UDDI, Web Services Description Language (WSDL), and Web services are focused primarily on the definition of provider-consumer or client-server interaction between entities. These standards need to be examined in the context of autonomic personal computing systems and extended to support new types of interactions, knowledge representation, and the collaboration needed to accomplish peer-to-peer autonomic behavior.

The autonomic framework that we have described is both a user of standards (e.g., Web services) and a candidate for standardization itself. As a standard it would encourage innovation in autonomic managers and the relationships between them.

Although the roles of autonomic manager and managed element are asymmetric, in peer group autonomic behavior it is often hard to statically associate the roles of a client or a server such that the current standards apply. The relationships among entities are more dynamic and reciprocal and often evolve after discovery and negotiation. The JXTA[25] work is an attempt to develop a technology for generalized peer-to-peer interaction among devices.

Unfortunately, the Web services model is not symmetric; it defines a supplier and a client of the service. Initially, we expect that autonomic systems will be created through the interconnection of existing systems, augmented by local autonomic behavior. In this evolutionary paradigm system, components take pseudostatic roles as clients or servers. Eventually, however, peer roles should be supported by standards.

Personal systems are often mobile and occasionally disconnected, so any interface to a remote autonomic manager must support a disconnected mode of operation. Disconnection is not explicitly supported by current Web services standards. Current implementations of Web services also have a design point that is resource-intensive, rather than the resource-constrained environment of personal computing. We look forward to personal versions of application servers, databases, discovery protocols for and peer group implementations of the Web services registry, and, in general, Web services implementations compatible with the resources that are available on a single PC.

## Summary and conclusions

Autonomic personal computing represents an important and distinct part of the autonomic computing concept. It is important because of the long-term potential to significantly reduce the frustration level and improve the user experience of hundreds of millions of PC users. It represents the potential to significantly reduce the personal computing support costs for millions of businesses. It is distinct because PCs exist in a more variable and less secure applications and connectivity environment than do servers. Personal computers are generally managed by less skilled personnel than are servers, centralized storage elements, or network infrastructure products.

Given the complex and dynamic environment faced by PCs (particularly mobile computers), we believe that an autonomic personal computing solution should adaptively seek out and leverage local, peer, and network resources. Sometimes the personal computing system will be disconnected from any net-

works and must be completely self-reliant. In other cases, the system should automatically communicate with and leverage the resources of its peers. When full network connectivity is available, the autonomic personal computing system may choose to take advantage of services provided over the network.

Personal computing requires very careful attention to privacy and security issues. To minimize exposure to attacks and to ensure security and privacy, we believe that security approaches based solely on software are insufficient, and approaches should take advantage of tamper-resistant hardware elements (e.g., TCPA) wherever possible. Furthermore, all communication and decision-making activities related to autonomic functions for each autonomic element should be coordinated by a trusted and secure autonomic manager function associated (perhaps dynamically) with each element.

Architectures and frameworks for autonomic personal computing must be based on open standards. Autonomic personal computing should also leverage technologies and approaches such as virtualization, peer-to-peer middleware, and Web services that are currently being applied in other problem domains. But the extensive interoperability, security, and ease-of-use requirements for autonomic personal computing will require new technologies and standards to be developed.

Autonomic personal computing represents a journey rather than a destination. The journey started long ago with the introduction of features such as plug-and-play and Dynamic Host Configuration Protocol (DHCP), and it continues today with improved backup, system recovery, and network-based management tools. It will continue with further improvements to server-based monitoring and management, peer-to-peer collaboration and information sharing, and more robust stand-alone diagnostic and recovery capabilities. But the greatest progress will occur only if an open architecture for autonomic personal computing is developed to enable secure, private, transparent, and adaptive collaboration between each personal computer and its dynamic environment.

## Acknowledgments

We thank those at the IBM Thomas J. Watson and Almaden Research Centers who participated in the autonomic personal computing brainstorming sessions of 2002. William Tetzlaff and Ed Lassettre were responsible for the autonomic manager concept. Special thanks to Alan Ganek, Kazuo Iwano, and William Chung for their perspectives on the relationship between autonomic personal computing and autonomic computing in general. We are indebted to the reviewers for their incisive and constructive comments.



## Cited references and notes

1. D. S. Siewiorek and R. S. Swarz, *Reliable Computer Systems: Design and Evaluation*, A. K. Peters Ltd., Natick MA (1998).
2. *The Journal of Computer Resource Management*, Computer Measurement Group, http://www.cmg.org.
3. R. J. Anderson, *Security Engineering: A Guide to Building Dependable Distributed Systems*, John Wiley & Sons, Inc., New York (2001).
4. CIM is standardized by the Desktop Management Task Force, http://www.dmtf.org.
5. *Microsoft Windows XP Professional Resource Kit Documentation*, Microsoft Press, Redmond, WA (2001).
6. Gnutella, at http://gnutella.org/, is just one example of file sharing.
7. R. J. Bayardo, R. Agrawal, D. Gruhl, and A. Somani, "YouServ: A Web-Hosting and Content Sharing Tool for the Masses," *11th International World Wide Web Conference*, Honolulu, HI (May 7–11, 2002), paper available at http://www.almaden.ibm.com/cs/people/bayardo/ps/www2002.pdf.
8. I. Foster, C. Kesselman, and S. Tuecke, "The Anatomy of the Grid—Enabling Scalable Virtual Organizations," http://www.globus.org/research/papers.html.
9. Web services activity of W3C, documented at http://www.w3.org/2002/ws/.
10. A. G. Ganek, "The Dawning of the Autonomic Computing Era," *IBM Systems Journal* **42**, No. 1, 5–18 (2003, this issue).
11. Windows Management Instrumentation, Microsoft Corporation, Redmond, WA, http://www.microsoft.com/hwdev/driver/WMI.
12. Different AMs may be in control of independent aspects of the resource; for example, one AM may control security, whereas another controls power consumption.
13. TCPA—The Trusted Computing Platform Alliance, http://www.trustedpc.org/.
14. S. Kent and R. Atkinson, "Security Architecture for the Internet Protocol," RFC 2401, Internet Engineering Task Force, available at http://www.ietf.org/rfc/rfc2401.txt.
15. A. O. Freier, P. Karlton, and P. C. Kocher, "The SSL Protocol" (March 1996), available at http://www.netscape.com/eng/ssl3/ssl-toc.html.
16. T. Dierks and C. Allen, "The TLS Protocol," RFC 2246, Internet Engineering Task Force (January 1999), available at http://www.ietf.org/rfc/rfc2246.txt.
17. A. Whitten and J. D. Tygar, "Why Johnny Can't Encrypt: A Usability Evaluation of PGP 5.0," Carnegie Mellon University, Pittsburgh, PA, available at http://www-2.cs.cmu.edu/~alma/johnny.pdf.
18. N. Borisov, I. Goldberg, and D. Wagner, "Intercepting Mobile Communications: The Insecurity of 802.11," *Seventh International Conference on Mobile Computing and Networking*, Rome (July 2001).

19. *Distributed Systems*, S. Mullender, Editor, Addison-Wesley Publishing Co., Reading, MA (1993).
20. D. Tapper, "Is e-Sourcing IBM Global Services' Formula for Dominating the Computing Utility Market?" International Data Corporation, Framingham, MA (2001).
21. D. Bantz, A. Mohindra, and D. Shea, "The Emerging Model of Subscription Computing," *IT Professional* **4**, No. 4, 27–32 (July/August 2002).
22. D. M. Russell, P. P. Maglio, R. Dordick, and C. Neti, "Dealing with Ghosts: Managing the User Experience of Autonomic Computing," *IBM Systems Journal* **42**, No. 1, 177–188 (2003, this issue).
23. S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, Prentice Hall, Upper Saddle River, NJ (1995).
24. R. P. Goldberg, "Survey of Virtual Machine Research," *Computer* **7**, No. 6, 34–45 (1974).
25. L. Gong, "Project JXTA: A Technology Overview," Project Juxtapose, available at http://www.jxta.org/project/www/docs/TechOverview.pdf.

*Accepted for publication August 28, 2002.*

**David F. Bantz** *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (electronic mail: bantz@watson.ibm.com).* Dr. Bantz has been a research staff member since 1972, after a short stint at a startup company. He graduated from Columbia University in 1970 with an Eng.Sc.D. degree and taught there as an adjunct professor for nearly 25 years. He has 20 issued patents. His technical interests have always been in personal computing applications and technology, and he is currently working on autonomic personal computing.

**Chatschik Bisdikian** *IBM Research Division, Thomas J. Watson Research Center, 19 Skyline Drive, Hawthorne, New York 10532 (electronic mail: bisdik@us.ibm.com).* Dr. Bisdikian is a research staff member, currently working on short-range wireless networks, wireless LAN deployment, service discovery, spontaneous networking, and peer networking. He holds a Ph.D. degree in electrical engineering from the University of Connecticut. He has served as vice-chair of the IEEE 802.15.1 task group that developed a standard for personal area networks adapted from the Bluetooth specification. He is coauthor of the book *Bluetooth Revealed: The Insider's Guide to an Open Specification for Global Wireless Communications*.

**David Challener** *IBM Personal Computing Division, 3039 Cornwallis Road, Research Triangle Park, North Carolina 27709 (electronic mail: challene@us.ibm.com).* Dr. Challener received his Ph.D. in applied mathematics from the University of Illinois, Champaign-Urbana and came to work for IBM upon graduation. Since then he has held positions in semiconductor yield analysis, substrate reliability, the technical staff to the president of the IBM PC Company, the Center for Natural Computing, PC architecture, and now the Personal Systems Institute, where he works on security and autonomic computing as a Senior Technical Staff Member.

**John P. Karidis** *IBM Personal Computing Division, Route 100, Somers, New York 10589 (electronic mail: karidis@us.ibm.com).* Dr. Karidis is an IBM Distinguished Engineer, developing hardware and software product concepts that have included the "butterfly" keyboard on the ThinkPad® 701C, now in the permanent collection of the Museum of Modern Art in New York, and the ThinkPad TransNote portfolio computer. He received his Ph.D. in mechanical engineering from The Pennsylvania State University and joined the Watson Research Center in 1983.

**Steve Mastrianni** *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (electronic mail: stevemas@us.ibm.com).* Dr. Mastrianni is a senior software engineer currently writing autonomic software. He joined IBM Research after running a software development and consulting company for 10 years. He has authored two books, several papers, and over 70 articles, and holds a Ph.D. in computer science. He has filed over 35 patents with six issued, and he prefers writing software to talking about it.

**Ajay Mohindra** *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (electronic mail: ajaym@us.ibm.com).* Dr. Mohindra has been a research staff member at IBM since 1993. He holds a Ph.D. in computer science from the Georgia Institute of Technology. His research interests include distributed systems and autonomic and e-utility computing.

**Dennis G. Shea** *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (electronic mail: dgshea@us.ibm.com).* Dr. Shea is the senior manager of the Personal Systems and Services department. He holds a Ph.D. in computer science from the University of Pennsylvania and electrical engineering degrees from Rensselaer Polytechnic Institute. He started his career at IBM in Boca Raton, Florida, working on small systems. His technical interests have revolved around personal systems technology, from easier connectivity and mobile solution enablement to the recent development of a desktop management computing utility.

**Michael Vanover** *IBM Personal Computing Division, 3039 Cornwallis Road, Research Triangle Park, North Carolina 27709 (electronic mail: vanover@us.ibm.com).* Mr. Vanover has been a development engineer since 1986. He graduated from Wheaton College in 1982 with a B.S. in chemistry. He later obtained a master's degree in electrical and computer engineering from the University of Texas at Austin. He has nine issued patents. His technical interests have ranged from processor design to graphics and multimedia, and more recently PC manageability and security.

# Clockwork: A new movement in autonomic systems

by L. W. Russell
S. P. Morgan
E. G. Chron

Statically tuned computing systems may perform poorly when running time-varying workloads. Current work on autonomic tuning largely involves reactive autonomicity, based on feedback control. This paper identifies a new way of thinking about autonomic tuning, that is, predictive autonomicity, based on feedforward control. A general method, called Clockwork, for constructing predictive autonomic systems is proposed. The method is based on statistical modeling, tracking, and forecasting techniques borrowed from econometrics. Systems employing the method detect and subsequently forecast cyclic variations in load, estimate the impact on future performance, and use these data to self-tune, dynamically, in anticipation of need. The paper describes a prototype network-attached storage system that was built using Clockwork, demonstrating the feasibility of the method, and presents key performance measurements of the prototype, demonstrating the practicality of the methods.

Large computing systems, especially those running time-varying workloads, are difficult to keep tuned. Dozens of interacting parameters may need to be understood and adjusted. Even if a system is tuned well at one point, because of changing workloads it may end up being poorly tuned at some other point. Badly tuned systems not only perform poorly, they also waste resources and frustrate users.

There is substantial and growing interest in autonomic systems, that is, systems that dynamically self-regulate. A key aspect of self-regulation is self-tuning. Current work on autonomic tuning is only slightly more advanced than static tuning; largely, such work revolves around primitive notions of *reactive autonomicity*, based on feedback control. Reactive autonomic systems reconfigure on the basis of instantaneous need or, at best, on the basis of short-term historical measurements. As with any techniques involving feedback control, reactive autonomic systems carry with them the well-known problems of potential instability or slow response to change.

In the next section of this paper, we propose a new approach to the problem. We introduce the concept of *predictive autonomicity*, based on feedforward control. We outline a general method, which we call *Clockwork*, for constructing predictive autonomically tuned systems. Using statistical modeling, tracking, and forecasting techniques borrowed from econometrics, systems employing the Clockwork method detect and forecast cyclic variations in their load, estimate the impact of the variations on future performance, and use these data to reconfigure themselves, in anticipation of need.

The third section describes a prototype, scalable network attached storage (NAS) system that we built using Clockwork, demonstrating the feasibility of the method. A *network attached store* is a network file server that processes requests sent to it using a pro-

Table 1 The Clockwork method

| Step | Electric Utility | Retail Chain | NAS Plex |
|---|---|---|---|
| 1. Establish system objective | Reliability (by rate class) | In-stock ratio (by sales class) | Response time (by file or client class) |
| 2. Establish measure of demand | Electricity, as it is being consumed | Sales, as they are being made | Requests, as they are being processed |
| 3. Track objective with demand | Reliability, as electricity is being consumed<br>Generator spin-up times<br>Instantaneous capacity | In-stock ratio, as items are being sold<br>Product distribution times<br>— | Response time, as requests are being processed<br>—<br>— |
| 4. Forecast demand | Use autoregressive time series analysis | | |
| 5. Adjust controllable parameters | Buy or sell electricity or options to buy or sell<br>Bring generators on or off line<br>Activate or deactivate spinning reserve | Issue store orders to distribution centers<br>Issue purchase orders to vendors<br>Liquidate excess inventory | Assign files to stores<br>Copy or move files between stores<br>Bring stores on or off line |

tocol such as Network File System (NFS),[1] over a medium such as Ethernet, by one or more clients. NFS, layered in turn on the Transmission Control Protocol/Internet Protocol (TCP/IP) suite, uses a remote procedure call architecture, in which every request from a client to a server engenders a response from the server to the client. Typical NFS requests are to create a file, to write data to a file, to read data from a file, and to delete a file. A response indicates whether the corresponding request was processed without error and, if so, contains request-specific data, for example, file contents from a read.

An NAS acts as a central repository for data shared among clients. With an NAS, clients need not each store the data, reducing cost. Clients need not coordinate updates to the data, simplifying their workings. Data management may be centralized, simplifying management and reducing costs. Small computers may be deployed widely; alternatively, large systems may be scaled further. It is desirable to have a powerful NAS to support more clients or to process more work from the same number of clients. For this paper, we prototyped one with a scalable architecture, integrating multiple stores into a single, virtual NAS. Requests are sent to the virtual NAS and are spread among the individual stores. The advantage of the architecture is that systems of various capabilities, including a very powerful system, may be built from relatively inexpensive components. The disadvantage is that the overall performance of a system will be only as good as that of its worst performing store. Although a virtual NAS could be massively overprovisioned to minimize the effect of one poorly performing store, that would reduce the advantage of the architecture. Alternatively, autonomic tuning could be used to balance load among the stores. We chose the latter approach.

Key performance measurements of the NAS prototype, demonstrating the practicality of the method, are presented in the fourth section. Finally, in the fifth section, directions for future work are suggested.

## The Clockwork method

Clockwork is a general method, analogous to those already in wide industrial use by electric power utilities and retail chains, for example. It enables a predictive autonomic system to be implemented following five simple steps, summarized in Table 1. The first two are configuration steps. They establish a system objective and a means to track it with load. The remaining three are operational steps. They automatically and continually track, forecast, and control the system.

A system that cannot be measured cannot be managed. Clockwork first establishes a simple, quantifiable objective, comprising a performance objective and a confidence level. For an electric utility, an appropriate performance objective would be to meet the instantaneous demand for electricity reliably. A potential performance objective for a retail chain

would be to achieve a certain in-stock ratio, a measure of how much product is in stock at a given time. For an NAS, achieving a certain average response time would be a suitable performance objective. The confidence level measures how closely the system must meet its performance objective. For example, the electric utility might need to meet demand 99.99999 percent of the time, the retail chain might need to achieve the in-stock ratio 90 percent of the time, and the NAS might need to achieve the average response time 66 percent of the time.

Often, objectives are subclassed. Some electric utility customers may be willing to trade decreased reliability for lower cost, some retail chains may require tighter controls on in-stock ratios for more profitable products, and some NAS clients may be willing to trade increased response time for lower cost. Although for brevity, the present discussion ignores subclassed objectives, Clockwork can handle them.

Clockwork, in the second step, establishes a simple, quantifiable measure of demand. An appropriate measure for an electric utility would be electricity being consumed; for a retail chain it would be sales being made; and for an NAS, it would be requests being processed.

Tracking the objective (and its variance) in relation to demand is the third step. An electric utility would track how reliably it met electricity demand, the time it took (or would take) for generators to be spun-up, and instantaneous capacity, as electricity was being consumed; a retail chain would track product in-stock ratios and product distribution times, as sales were being made; and an NAS would track response time, as requests were being processed.

In the fourth step, demand is forecast, along with uncertainty, using autoregressive time series procedures. This technique projects future values of a variable based on the history of that variable alone, which simplifies forecasting considerably. A key contribution of Clockwork is that *the same procedure would be used by the utility, the retail chain, and the NAS*.

Fifth and finally, the controllable parameters of the system are adjusted to meet the objective. In anticipation of forecast demand: the electric utility would bring its generators on or off line, would buy or sell electricity or options to do the same, or would activate or deactivate its spinning reserve; the retail chain would issue store orders to its distribution centers, would issue purchase orders to its vendors, or would liquidate its excess inventory; and the NAS would reassign files to stores, would replicate files among or migrate files between stores, or would bring stores on or off line.

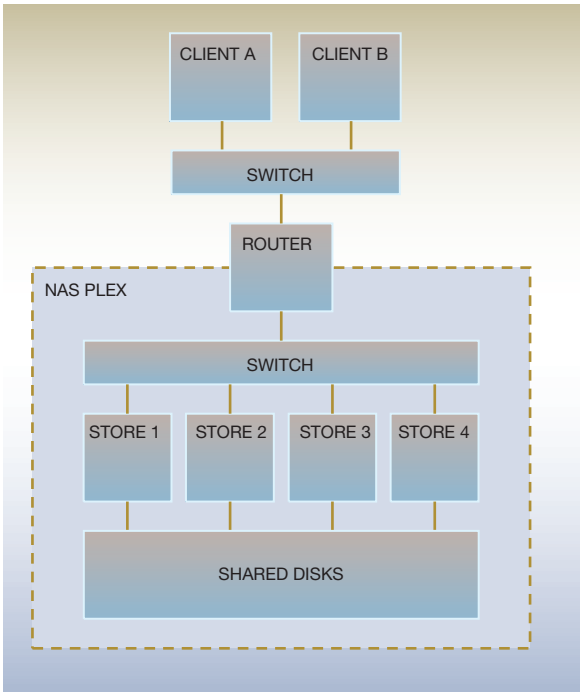
## The prototype

In this section, we describe how we used Clockwork to prototype a scalable, autonomically tuned NAS. Our purpose in building the prototype was to determine whether the method is feasible and practicable, rather than to achieve optimal performance. Nevertheless, as the measurements in the next section show, the prototype performs well. For proof of concept, and because we were able to operate in a shared-disk environment, we implemented file reassignment, but not file replication (copying a file to multiple stores) or migration (moving a file between stores). Had we been faced with a serially shared disk or a shared-nothing environment, we would have had to have implemented replication and migration.

The prototype comprises three main components: a set of stores, or storage servers, that process requests for files kept in a cluster file system, a request router that spreads requests among the stores, and an autonomic control program that directs the router, following the Clockwork method. We call the overall system an *NAS plex*, as it integrates multiple, otherwise independent systems. The prototype NAS plex is depicted within the dashed-line area of Figure 1. It includes four stores, a router, an internal network, and shared disks. Two clients are connected to the plex via an external network.

The clients, the router, and the stores are computers with an Intel architecture. With the exception of the router, all computers run the Linux** operating system. The router runs a real-time operating system to minimize latency and runs the Clockwork control program. The stores share files via the General Parallel File System (GPFS)[2] cluster file system, which manages fibre channel disks. The prototype is interconnected via Fast Ethernet. Clients access files via NFSv3/UDP (Network File System version 3/User Datagram Protocol). Although the clients are configured identically, the stores deliberately are not, so that the prototype is inherently unbalanced (see below). The stores contain processors of various speeds. Some stores have one processor, whereas others have two. Stores have different amounts of memory. We used GPFS[2] because it is a robust IBM product that supports the hardware and software

Figure 1 The prototype NAS plex, as built




used in the prototype.[3] GPFS implements a scalable, shared disk architecture. Although the prototype used GPFS, the IBM Storage Tank*[4] storage area network (SAN) file system was a viable alternative.[5]

The prototype works as follows. A client sends a request to the router, which forwards it to a store for processing. Any store may access any file, since files are managed by a cluster file system, which coordinates accesses to them. Which store will process a given request is a decision made by Clockwork based on the type of the request, the file to which it refers, and the state of Clockwork. The decision process is described in more detail below. The architecture enables the load of the NAS plex to be shared among its stores.[6] Load balancing, or intelligently sharing load, has two main benefits. First, as with any modern computer system, performance is nonlinear. Past a saturation point, a linear increase in load causes a much greater increase in response time. Load balancing can keep the plex operating within a linear performance region. Second, assigning related requests to the same store can take advantage of data caching, thereby keeping the number of I/Os, and the amount of computation, low.

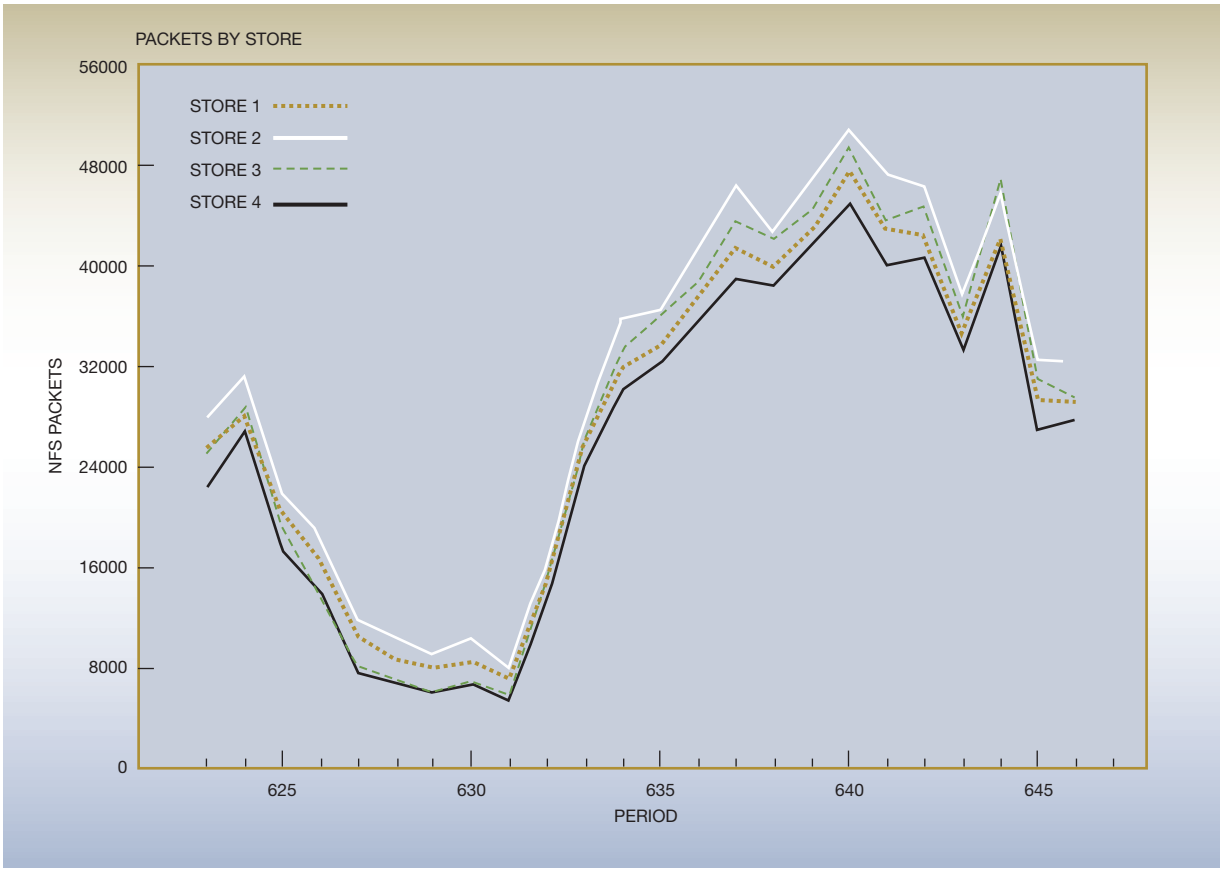
For this architecture, load could be balanced statically, that is, files could be assigned to stores following a fixed schedule, or it could be balanced dynamically, with file assignment changing over time. In reality, static assignment would prove a poor choice. Requests arriving clustered in time tend to be related, load tends to include multiple cyclical components, and load tends to vary substantially over time. The prototype is dynamically balanced using feedforward control.

The router is NFS request- and response-aware. It analyzes and routes requests and responses at network speeds. The router records statistics on a per-file, per-request basis, as well as on a per-store, per-response basis. It forwards requests to appropriate stores using a default rule and an exception set. The default rule—the prototype uses a simple hash of the NFS file handle (or file identifier) to choose a store—has several characteristics. It spreads load more or less uniformly among the stores. It repeatably assigns a given file to the same store. It is simple to compute. Given these characteristics, the rule takes advantage of store data (and cluster file system token) caching; however, it assigns files to stores statically, ignoring store load and file heat, that is, the extent to which the file contributes to load.

Using the statistical data gathered by the router, the Clockwork control program periodically: tracks and forecasts store response time at a given load; tracks and projects per-file heat; estimates the effect on response time of reassigning hot files to stores; decides which files to reassign, and updates the exception set of the router to reassign the files. On the assumption that there are cyclical components to access patterns, the projections and assignments of Clockwork are refined over time, as its statistical database grows. Clockwork detects and adjusts rapidly to any fundamental changes in access patterns.

Clockwork projects the expected load of each file using well-known time series analysis methods borrowed from econometrics. In particular, Clockwork models load using an Autoregressive Integrated Moving Average (ARIMA) model[7] from which it extracts cyclical components. Clockwork applies Geweke's Spectral Forecasting procedure[8] to the components to forecast future load from present load. In essence, the number of requests per period is viewed as an infinite moving average, a Fourier transform of the time series is estimated, a corresponding time-domain model is computed, and the model is used to forecast load. As the same model

Figure 2 Default distribution of NFS requests, by store




applies to all such series, the procedure can be automated.

Using the load forecast, Clockwork determines which stores, if any, are likely to be overloaded in the next period. It iteratively proposes a reassignment of files from overloaded to underloaded stores. Files are proposed for reassignment in descending order of heat. Iteration terminates when the performance objective of the plex is achieved or, if the objective cannot be achieved because of plex overload, when the load is balanced.

Given a proposed assignment, Clockwork estimates the response time of a store using Hannan's Efficient Estimator,[9] a spectral procedure for estimating generalized least squares. This procedure is applicable assuming that all factors taken together, other than the number of requests processed in a period, follow a stationary ARIMA process. In practice, this assumption has proven reasonable. Because the same model applies to all data series, the procedure again can be automated.

Rather than use a default rule and iteratively proposing reassignment of a few hot files, the prototype could have computed an optimal assignment following a stochastic optimization model, with Benders decomposition and Lagrangian relaxation. See Dentcheva and Romisch[10] for examples of such computations. The model is completely specified, both from a mathematical standpoint and in terms of statistical estimation procedures. The procedures require historical data on load and response time, which the router gathers and records. In reality, such a computation would be highly complex and slow. Given the existence of a simple default rule, Clockwork has an adequate starting point from which to

Figure 3 Measured average response time by store, without reassignment

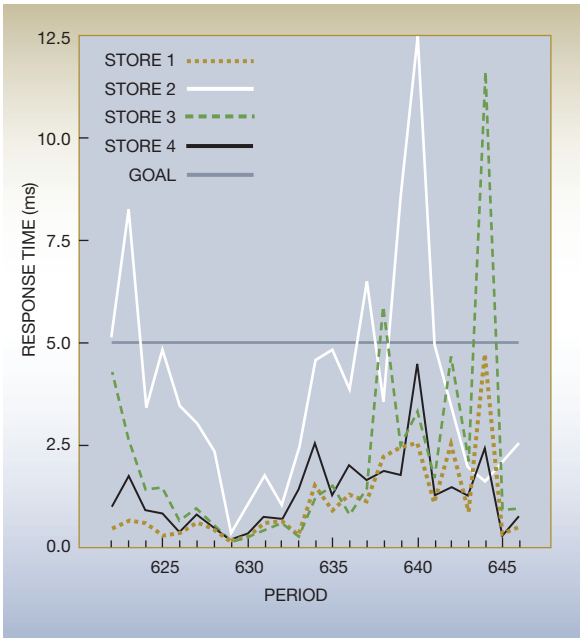
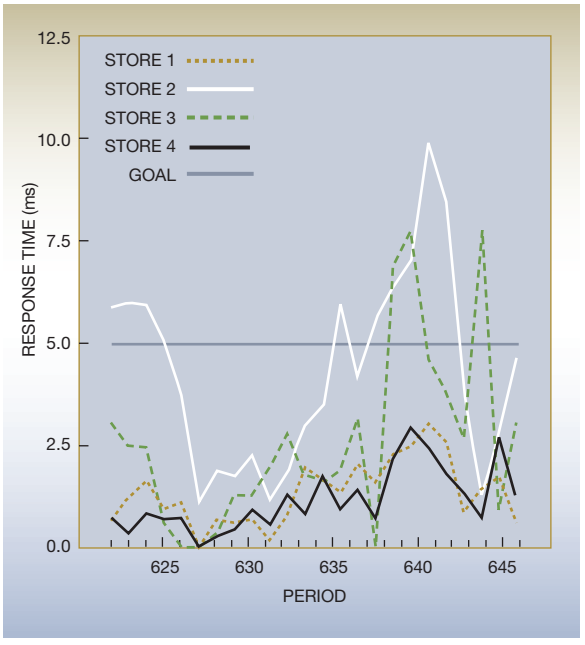



Figure 4 Forecast average response time by store, without reassignment




iteratively apply incremental changes, which quickly leads to good results. There is no need to apply a more complex procedure.

## Measured results

There are no generally accepted, long-running NFS traces suitable for evaluating the prototype. For predictive systems, synthetic workloads are inappropriate, because they invariably contain artificial cycles or are highly random, leading either to perfect results or perfectly useless ones. Lacking real workloads, yet desiring independently reproducible results, we generated NFS workloads from four essentially different HyperText Transfer Protocol (HTTP) traces we downloaded from HitBox.[11] We chose those from a "fantasy" soccer site on which users create virtual teams with which they play virtual matches, a memorabilia site on which users trade sports and other memorabilia, a name definition site, which expectant parents use to choose a name for their baby, and an MP3 download site.

We used Fstress,[12] an NFS benchmarking tool, to generate the actual workloads from the HTTP traces. For each, we constructed an appropriate set of files, numbering over 1100, total. We determined a base load, at which all four workloads were issuing requests at a heavy rate, and under which the plex was stressed; that is, its response time was changing nonlinearly as a function of load. We evaluated the system with the workloads running simultaneously and independently. First, we ran the workloads with file reassignment disabled, and again with it enabled. The results given below correspond to a representative 24 hours of the trace, starting 622 hours into it.[13] Each period corresponds to one hour of the trace.

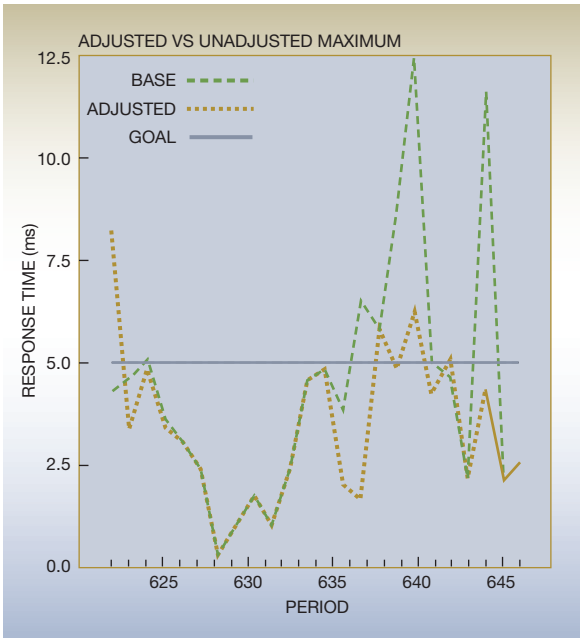
Figure 2 shows the distribution of NFS requests by store following the default rule. The rule tends to spread requests evenly among the stores. Figure 3 shows the measured average response time, by store, without file reassignment. Clearly, the stores perform very differently. Figure 4 shows a forecast average response time, by store. In a comparison of Figures 3 and 4, the projections seem acceptably close. Notably, Clockwork detects the differences among the stores and projects them forward.

Next, we reran the traces through the prototype with the file reassignment of Clockwork enabled, and with an appropriate objective: to achieve a 5 ms or better average response time at a 66 percent confidence level. The goal was chosen to demonstrate the prac-

ticality of the method, not to achieve the best possible results. Other goals could have been chosen that also would have demonstrated practicality, for example, reducing average response time by 10 percent. With the chosen goal, files were reassigned in 11 of the 24 periods. In 10 periods, files were reassigned from Store 2; in three periods, files were reassigned from Store 3; and in two periods, files were reassigned from both Stores 2 and 3. In all cases, the files were reassigned to Store 4, the best performing store.

Figure 5 shows the effect of the reassignments on response time. The graphs show the maximum of the average per-store response times, where base indicates the measured times without reassignment, and adjusted indicates the times with reassignment. The prototype achieved the performance component of its goal, or came very close, in nearly all periods. It missed by more than the calibration error only in periods 637 and 639. Given the confidence level chosen, it achieved its overall goal. It is notable that, during Periods 639 and 643, including one of the periods in which it missed its performance goal, the prototype shaved the maximum average response time of the plex nearly in half.

Figure 5 Measured maximum average response time, with and without reassignment




## Future work

We are continuing this work in several areas. We have extended the router to translate incoming NFS/TCP connections to NFS/UDP inside the plex to balance a connection-oriented NAS protocol. We have projected per-file workloads multiple periods out, with encouraging results. Given the multiperiod results, we believe it will be possible to balance load by file replication and migration, extending the method to serially shared-disk and shared-nothing environments.

Re-examining Figure 5, we see that the prototype incorrectly forecast an overload in periods 624 and 641, which led to slightly worse response times. Although we argue against feedback control as the sole method for autonomic tuning, integrating some form of feedback control with Clockwork may improve the method. In the noted periods, a real-time monitor could have detected that actual load was deviating from the forecast, and might temporarily have overridden, or perhaps canceled, reassignment. It is unclear what steps should be taken in general. See Wang and Morris[14] for a comprehensive study of load monitoring and balancing techniques, including some that may be appropriate for integration with Clockwork.

## Conclusions

In this paper, we proposed a new approach to autonomic systems. We introduced the concept of a predictive autonomic system, which regulates its behavior in anticipation of need, using statistical modeling, tracking, and forecasting procedures. We proposed the Clockwork method for autonomic systems. We demonstrated the feasibility of the method, using it to prototype a self-tuning NAS plex. We presented measurements of the prototype under substantial workloads. The measurements demonstrate the practicality of the method. Finally, we discussed future work.

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of Linus Torvalds.

## Cited references and notes

1. B. Callaghan, B. Pawlowski, and P. Staubach, *NFS Version 3 Protocol Specification*, Request for Comments 1813, Internet Engineering Task Force, Mountain View, CA (June 1995).

2. F. Schmuck and R. Haskin, "GPFS: A Shared-Disk File System for Large Computing Clusters," *Proceedings of the FAST 2002 Conference on File and Storage Technologies*, USENIX Association, Monterey, CA (January 2002), pp. 231–234.
3. GPFS also runs on IBM RISC System/6000® SP™ parallel systems.
4. Storage Systems Group. IBM Almaden Research Center, Storage Tank Web page, see http://www.almaden.ibm.com/cs/storagesystems/stortank/.
5. GPFS and Storage Tank originated at the IBM Almaden Research Center, where the present work also was done.
6. The architecture can also conceal certain system administrative tasks such as adding, removing, or upgrading a store, and can mask certain hardware or software failures.
7. G. E. P. Box, G. M. Jenkins, and G. C. Reinsel, *Time Series Analysis, Forecasting and Control,* Third Edition, Prentice-Hall, Upper Saddle River, NJ (1994).
8. J. Geweke, *Priors for Macroeconomic Time Series and Their Application*, Discussion Paper No. 64, Federal Reserve Bank of Minneapolis, Institute for Empirical Macroeconomics, Minneapolis, MN (1992).
9. E. J. Hannan, "Regression for Time Series," in *Time Series Analysis*, M. Rosenblatt, Editor, John Wiley & Sons, Inc., New York (1963), pp. 17–37.
10. D. Dentcheva and W. Romisch, "Optimal Power Generation Under Uncertainty via Stochastic Programming," in *Stochastic Programming: Numerical Techniques and Engineering Applications*, K. Marti and P. Kall, Editors, *Lecture Notes in Economics and Mathematical Systems* **458**, 22–56, Springer-Verlag, Berlin (1997).
11. HitBox, WebSideStory, San Diego, CA, available from http://www.hitbox.com/.
12. D. C. Anderson and J. S. Chase, *Fstress: A Flexible Network File Service Benchmark*, in preparation; available from Duke University, Department of Computer Science at http://www.cs.duke.edu/ari/fstress/download/paper.pdf.
13. There is no significance to this particular choice of starting period.
14. Y.-T. Wang and R. J. T. Morris, "Load Sharing in Distributed Systems," in *ACM Transactions on Computing Systems* **34**, No. 3, 204–217 (March 1985).

*Accepted for publication October 10, 2002.*

**Lance W. Russell** *IBM Research Division, Almaden Research Center, 650 Harry Road, San Jose, California 95120 (electronic mail: lancerus@almaden.ibm.com).* Mr. Russell is a research staff member in the Storage Systems Group at the Almaden Research Center. While at the IBM Advanced Workstations Division, he led the early AIX® operating system architecture and implementation efforts for TCP/IP, sockets, streams, and netware. He received an IBM Outstanding Technical Achievement Award for his work on AIX 3.1. At Hewlett-Packard Laboratories, he worked as a principal scientist on server blade architecture and dynamic resource allocation. He holds over two dozen patents and has numerous patent filings. In addition to systems architecture, his research interests include forecasting and system planning under uncertainty. He has testified on the latter subjects before the Public Service Commission of Colorado and the Federal Energy Regulatory Commission.

**Stephen P. Morgan** *IBM Research Division, Almaden Research Center, 650 Harry Road, San Jose, California 95120 (electronic mail: smorgan@almaden.ibm.com).* Mr. Morgan is a senior software engineer in the Storage Systems Group at the Almaden Research Center. He has worked on a variety of systems projects since joining the IBM Thomas J. Watson Research Center in 1981. He was a member of the 801 Group and helped transfer its technology into the AIX 3.1 operating system. He is a coinventor of the AIX Logical Volume Manager. Mr. Morgan helped jump-start the Open Software Foundation. He developed High Availability for NFS. He was principal investigator on a Department of Defense secure workstation project, for which he received an IBM Outstanding Innovation Award. He holds more than a dozen patents and has published papers at conferences ranging from VLDB to USENIX. He is primarily interested in storage systems and computer networking. In 1982, Mr. Morgan received an S.B. degree in computer science and engineering from the Massachusetts Institute of Technology.

**Edward G. Chron** *IBM Research Division, Almaden Research Center, 650 Harry Road, San Jose, California 95120 (electronic mail: echron@almaden.ibm.com).* Mr. Chron is a senior software engineer in the Storage Systems Group at the Almaden Research Center. He worked on hardware architecture-specific operating system technology and system software development, and on CAD and CAE (computer-aided design and computer-aided engineering) tools for chip layout and design, while at Intel Corporation. At the former IBM Palo Alto Scientific Center, which he joined in 1991, Mr. Chron developed advanced versions of the AIX operating system and of the Mach microkernel, for the IBM RS/6000 and for IBM mainframe systems. Since moving to the Almaden Research Center in 1993, Mr. Chron has worked on numerous systems projects. His research interests include operating systems, systems architecture, distributed and networking systems, and storage-related applications. He received a B.S.E.E. degree from the University of Illinois, Urbana Champaign in 1981.

# Comparing autonomic and proactive computing

by R. Want
T. Pering
D. Tennenhouse

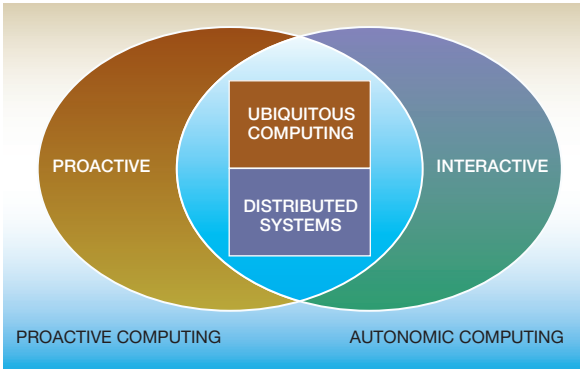
This paper provides an overview of the relationship between proactive computing and autonomic computing, considering the design of systems that are beyond the scope of our existing computational infrastructure. Autonomic computing, as described by IBM's manifesto on the subject, is a clear statement of the difficulties and challenges facing the computing industry today. In particular, autonomic computing addresses the problem of managing complexity. Intel Research is exploring computing futures that overlap autonomic computing but also explore new application domains that require principles we call proactive computing, enabling the transition from today's interactive systems to proactive environments that anticipate our needs and act on our behalf.

Autonomic[1] and proactive[2] computing both provide solutions to issues that limit the growth of today's computing systems. In the 1990s, the ubiquitous computing vision[3] extended what has been traditionally called distributed systems, a field in which the application focus has been primarily office automation.

To date, the natural growth path for systems has been in supporting technologies such as data storage density, processing capability, and per-user network bandwidth, with growth increasing annually for 20 years by roughly a factor of 2 (disk capacity), 1.6 (Moore's Law), and 1.3 (personal networking; modem to DSL [Digital Subscriber Line]), respectively. The usefulness of Internet and intranet networks has fueled the growth of computing applications and in turn the complexity of their administration. The IBM autonomic vision seeks to solve some of the problems from this complexity by using eight principles of system design to overcome current limitations. These principles include the ability of systems to self-monitor, self-heal, self-configure, and improve their performance. Furthermore, systems should be aware of their environment, defend against attack, communicate with use of open standards, and anticipate user actions. The design principles can be applied both to individual components and to systems as a whole, the latter providing a holistic benefit that satisfies a larger number of users.

At Intel Research we enthusiastically support the aims of autonomic systems and at the same time consider how computing systems will be used in the future. To date, the familiar personal computer (PC) infrastructure has been applied most effectively in the realm of the office and the home. Going forward, we are intrigued by other areas of human endeavor that are ripe for the application of computer-based technology. Proactive computing extends our horizon by recognizing a need to monitor and shape the physical world, targeting professions that have complex real-world interactions but are currently limited by the degree of human involvement required. We are addressing some of the challenges that exist beyond the scope of earlier ubiquitous computer systems to enable future environments involving thou-

Figure 1 The relationship of computing paradigms




sands of networked computers per person. Proactive system design is guided by seven underlying principles: connecting with the physical world, deep networking, macro-processing, dealing with uncertainty, anticipation, closing the control loop, and making systems personal.

An emphasis on human-supervised systems, rather than human-controlled or completely automatic systems, is an overarching theme within proactive computing. Computer-to-user ratios have been changing over time: 1:many turned into 1:1 with the advent of the PC in the 1980s, and into many:1 with the explosion of mobile devices in the new millennium. Currently, most people in the United States typically own (sometimes indirectly) many tens of computers, ranging from portable devices to consumer electronics. These systems compete for human attention, an increasingly scarce resource in modern living. Before the sheer number of devices overwhelms us, solutions need to be found to remove people from the control loop wherever possible, elevating their interaction to a supervisory role. One way would be to use pure artificial intelligence, a lofty goal that will not be attainable in the near future. Proactive computing, therefore, focuses on human-supervised operation, where the user stays out of the loop as much as possible until required to provide guidance in critical decisions.

A simple present-day example that illustrates a human-supervised system is a modern home central heating system. Such systems typically have a simple regime for morning, day, evening, and night temperature settings. Normally, the system operates untended and unnoticed; however, users can readily override these settings at any time if they feel hot or cold, or to address an impending energy crisis. Furthermore, if the system were instrumented with a sensor network and knowledge of a family's calendar, the temperature and energy consumption could be optimized proactively to allow for in-house microclimates, late workdays, and family vacations. However, extending this example to more complex systems is quite a challenge—most decisions do not simply become a selection between "too hot" or "too cold."

As illustrated in Figure 1, there is considerable intellectual overlap between research into autonomic and proactive systems. Both autonomic and proactive systems are necessary to provide us with tools to advance the design of computing systems in a wide range of new fields. In the following sections we provide an overview of the issues, technology directions, and examples of why both these visions are necessary.

## Extending the application domain

Enabling a computing future that goes beyond the current in-home and in-office application domains will require the adoption of new design principles. Here we examine three of the seven proactive principles: connecting with the physical world, real-time/closed-loop operation, and techniques that allow computers to anticipate user needs. Readers interested in the remaining four are directed to a description of proactive computing on the Web at http://www.intel.com/research.

**Connecting to the physical world.** Most of our computing infrastructure to date connects personal computers through networks to arrays of servers. The resulting systems provide us with a virtual environment allowing us to author, process, and file information, which, through people, can have an indirect influence on the physical world. To enable a world in which computing aids us in our day-to-day tasks, the physical world must be instrumented so that computer systems can have direct and intimate knowledge of our environment, ultimately using that information to effect change. Corresponding examples are microclimate weather forecasts, monitoring road traffic, and determining where people might be located in an earthquake-damaged building.

Needless to say, there are a number of inherent problems in building such a system. First, there are pragmatic issues such as maintenance, connectivity, and

finding suitable power supplies. Second, we are describing systems that, when applied on a city or national scale, have never before been built. The coordination and management issues take on a new level of difficulty; new protocols need to be created to enable appropriate data flow; and power management becomes a critical parameter for sensors that must operate from independent energy sources. Applying sensors to the physical world on a national or global scale is a daunting task, but as our societies become more complex and population densities increase, the payback will be worth it.

Scaling systems to a size large enough to monitor the physical world raises immediate problems of administration and utilization—the very problem that autonomic computing sets out to solve—and we cannot simply look to existing computer systems for guidance. However, by using simple nodes that can be individually and comprehensively characterized, it may be possible to learn more about the techniques required to maintain larger networks of conventional computers, informing both proactive and autonomic system builders. Multihop wireless sensor networks, such as the networks we are working on in collaboration with our colleagues at the University of California, Berkeley,[4,5] have exactly this characteristic.

**Real-time and closed loop operation.** If we expect our computers to become more integrated with the physical world, real-time response will become a critical factor that needs to be supported by all computer systems. In the 1960s, computer systems were either fully interactive, putting humans in the control loop, or completely inflexible, built on a dedicated control system. In order to integrate systems fully into real-world tasks, the systems must be able to respond faster than is possible with a person in the control loop: they must have real-time response to physical-world events.

If general-purpose computing systems were redesigned to make real-time guarantees, many new proactive applications would be possible and perhaps even begin to appear as mass-market shrink-wrapped software in major retail stores. However, the underlying issue is that most software systems make no guarantee of a real-time response, instead hiding behind layers of abstraction without considering the response time induced by varying conditions. Those of us familiar with the embedded systems world typically resort to specialized software based on real-time operating systems (RTOS) for critical control applications, capabilities that are not supported by most general platforms.

**Anticipation.** Anticipation is a cornerstone of proactive computing. For systems to be truly proactive, they need to in some sense predict the future. Our research is currently focusing on the use of context, statistical reasoning, and data-handling, all summarized below, as a baseline for anticipating a user's needs. Utilizing these techniques, and others, will allow systems to quickly handle real-world situations and provide the appropriate level of user interaction.

***Context aware operation.*** Portable and wirelessly connected systems have opened up the opportunity to use contextual information, such as physical location and the availability of surrounding infrastructure, to modify the behavior of applications. Both autonomic and proactive systems can take advantage of context by using the environment in which they operate to guide policy decisions. Autonomic computing can be of benefit directly in supporting new configurations, for example, through the local discovery of resources and setting up default operation. Proactive systems, working at a higher level, can filter information for display and customize the effects of commands.

Location is one of the most useful parameters to define context, and making high-fidelity location information available to mobile devices and their supporting systems is one of our immediate research goals. Some of our research programs are looking at ways to track the location of objects inside a building (beyond the capabilities of GPS, the Global Positioning System), taking advantage of the properties of existing wireless networks, or finding solutions for augmenting environments in a cost-effective way. We are also developing a location representation and application interface that allows common access to the data, essentially examining the type of protocol stack that might be useful as a standard to fuse and disseminate location information.

***Statistical reasoning.*** In the last decade there have been advances in analytical techniques that use statistical methods to solve important problems. These techniques have expanded and even replaced some of the more traditional approaches using deterministic methods. Examples of applied techniques include Hidden Markov Models, genetic algorithms, and Bayesian techniques. We believe there is considerable benefit in applying these techniques to the management and analysis of large systems, both in

the information technology field and for process control and manufacturing in industry.

In some World Wide Web applications such as the Google search engine, these techniques are already being applied to data mining. Other successful areas of computer science that use statistical techniques are speech recognition, vision processing, and even the routing algorithms used by some computer-aided design tools. Moving forward, we will apply statistics to information contained in the physical world on a real-time basis.

***Proactive data-handling.*** The exponential increase in the density of data-storage technology and the increasing network bandwidth available for data transport provide the means for proactive computing systems to quickly provide data to users without their explicit intervention. Proactive computing systems can take advantage of high-density portable storage that allows systems to prefetch data, which might be useful to users in the future without burdening them with a cumbersome mobile device. Likewise, high-bandwidth networks can move bulk data to a server physically near a user in a short period of time—a technique we call data staging. However, autonomic techniques must make certain that users are able to trust such systems by ensuring they are able to operate under a wide variety of conditions.

Both local data caching and data staging can play a vital role in supporting user mobility. Networks provide invaluable up-to-date connectivity, but if relied on completely, will sometimes fail a user when they are not available or become congested. Local data caching, in contrast, can serve a user if the cache contents are well chosen, but the data may not always be the latest version. By utilizing both of these techniques, proactive computing aims to make data available to computers moving through the physical world in real time, thus supporting the overall vision.

## Catalyzing research

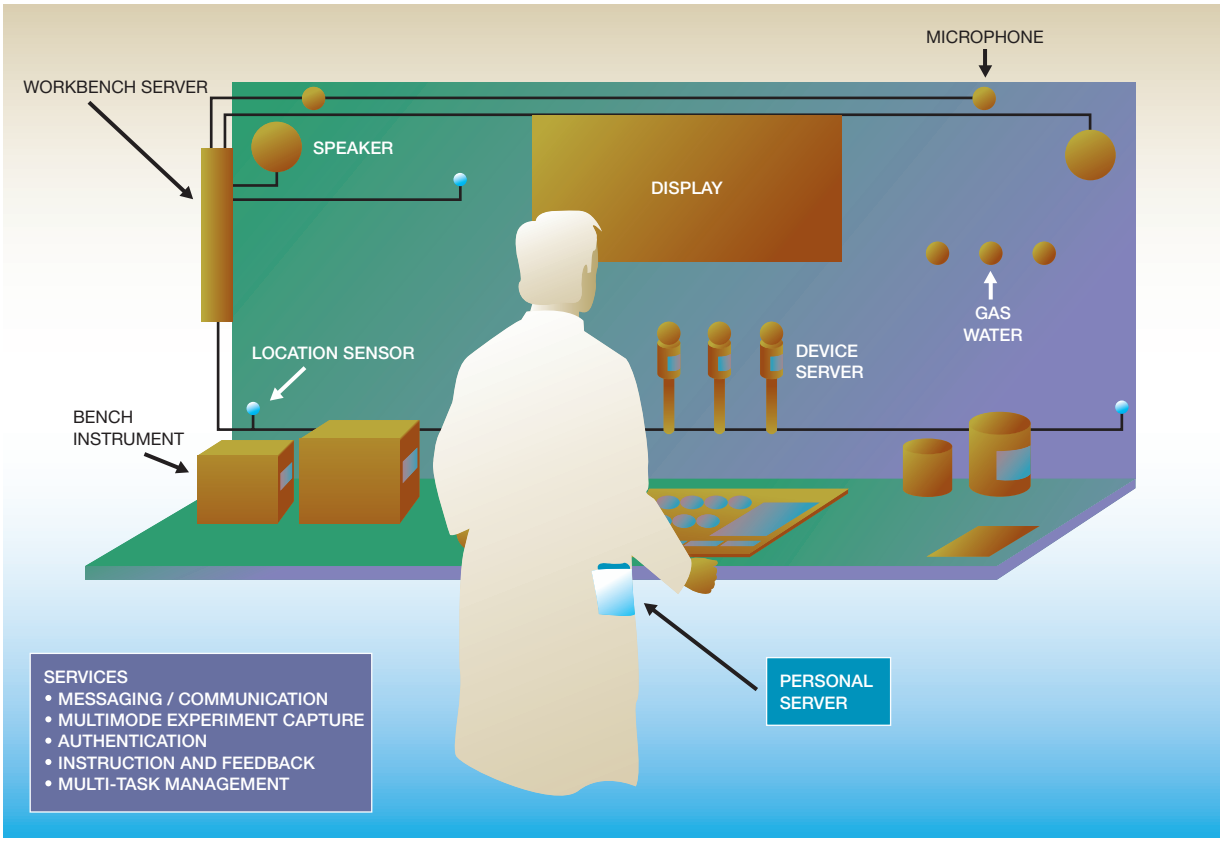
In order to embrace these challenges, we briefly summarize two of the projects that are part of Intel Research's project portfolio and designed to drive research and the use of computers beyond traditional environments.

**Labscape.** This project (in collaboration with the University of Washington) sets out to augment a microbiology laboratory and automate the recording and analysis of results, a prime example of an environment for which computing has had little impact. Labscape[6] sets out to instrument reagents, reaction vessels, test equipment, and the staff, and track their relative location during the experimental process (Figure 2). During any experimental procedure, many processes need to be recorded in a laboratory notebook; however, some steps are sometimes omitted, and sometimes, cross contamination occurs between reagents. In a traditional laboratory these failures can only be tracked down by skilled staff; there are no inherent mechanisms for monitoring status. With Labscape, the whole experiment can be recorded electronically and automatically generate a notebook entry for the method and results. The benefit is that no steps are accidentally lost, and furthermore, an expert system can examine the data for potential contamination risks and other experimental pitfalls.

In Labscape, a complex web of computation is created as the result of many communicating components. The principles of autonomic computing are essential as the underpinning for these systems, enabling the components to reliably and efficiently cooperate with one another. However, it is also beyond the scope of traditional computing environments touching the physical world, needing real-time response, and keeping the user out of the computational loop wherever possible. Thus proactive computing plays a vital role in the management and coordination of such a system, making inferences and using context to record data and assess risk.

**The Personal Server.** The Personal Server[7] focuses on a user's interaction with personal mobile data through the world around him or her, inspired by the trends in computation, storage, and short-range wireless communication standards. The underlying thesis is that storage density, which is doubling annually, will lead to one-inch disks that may store over one terabyte by 2012. With this information density available, it will be possible to carry vast amounts of data in one's pocket, some of which may really be needed, and other information that might be included, just in case. The device, which we call a Personal Server (Figure 3), can be small enough that a person will always have it available, perhaps embedded in a cellular phone or worn as jewelry. Because it does not rely on an integrated display as its primary interface, it can be quite small and still provide rich interaction. It is designed to take advantage of the surrounding computing and display infrastructure, allowing information to be opportunistically viewed on neighboring displays, thus free-

Figure 2 Labscape—instrumenting a microbiology laboratory (used with permission of Larry Arnstein, University of Washington)




ing users from carrying the bulk and weight of a screen. Standard wireless protocols, which typically provide the mechanism for sending data between the device and the host, can be used in an *ad hoc* and proactive way to discover useful information in the environment and record it for future use. Similar opportunities occur when a Personal Server encounters other Personal Servers that may advertise particular information, enabling personal peer-to-peer sharing.

Once again autonomic principles are required for this system to be successful, establishing a sense of self for the Personal Server and guarding against possible adverse data or programs that may be pushed onto it. In addition, proactive techniques are required for predicting the types of data the user needs, based on previous data access patterns or the context of the user.

Figure 3 A Personal Server prototype

## The future of computing

Computing has reached a point where conventional office-bound information technology is no longer the main driver for the expansion of computational infrastructure. There are many tasks, both exotic and mundane, that can benefit from applied computation. The networking of embedded computers will unlock data that are presently stranded and allow us to apply computation beyond traditional boundaries. As these data flow into larger systems, new opportunities will be found to bring about productivity gains from the data and to offer new services that impact our lives. However, dealing with thousands of processors per person, and the torrents of data that they provide, will force us to move from interactive to proactive paradigms. This move is the aim of the proactive computing program at Intel Research, which encompasses activities in universities and industry alike to develop mechanisms that support proactive behavior.

It is clear that many of the examples we have described will also rely on the principles of autonomic computing, because they have an inherent need for self-configuration, self-healing, and self-monitoring. These factors are necessary for scalable systems and thus are integral to both endeavors.

We have all enjoyed an exciting ride as the computing industry has moved faster than any other in history in terms of its technological progress, mainly because of the exponential growth factors driving higher processor performance, increasing memory density, and lowering power consumption. However, the ride is far from over, and under the auspices of autonomic and proactive techniques, computers will take us in some very new directions.

## Acknowledgments

We would like to acknowledge all members of Intel Research and our colleagues at IBM whose work focuses on turning the visions of autonomic and proactive computing into reality.

## Cited references

1. P. Horn, *Autonomic Computing*: *IBM's Perspective on the State of Information Technology*, IBM Corporation (October 15, 2001); available at http://www.research.ibm.com/autonomic/manifesto/autonomic_computing.pdf.
2. D. L. Tennenhouse, "Proactive Computing," *Communications of the ACM* **43**, No. 5, 43–50 (May 2000).
3. M. Weiser, "The Computer of the 21st Century," *Scientific American* **265**, No. 3, 94–104 (September 1991).
4. S. Conner, L. Krishnamurthy, and R. Want, "Making Everyday Life Easier Using Dense Sensor Networks," *Proceedings of Ubicomp 2001: 3rd International Conference on Ubiquitous Computing*, Atlanta, GA, Springer, Lecture Notes in Computer Science **2201** (October 2001), pp. 49–55.
5. D. Estrin, D. Culler, K. Pister, and G. Sukhatme, "Connecting the Physical World with Pervasive Networks," *IEEE Pervasive Computing* **1**, No. 1, 59–69 (January–March 2002).
6. L. Arnstein, C. Hung, R. Franza, Q. Zhou, A. LaMarca, G. Borriello, S. Consolvo, and J. Su, "Labscape: A Smart Environment for the Cell Biology Laboratory," *IEEE Pervasive Computing* **1**, No. 3, 13–21 (July–September 2002).
7. R. Want, T. Pering, G. Danneels, M. Kumar, M. Sundar, and J. Light, "The Personal Server: Changing the Way We Think about Ubiquitous Computing," *Proceedings of Ubicomp 2002: 4th International Conference on Ubiquitous Computing*, Goteborg, Sweden (September 30–October 2, 2002), Springer, Lecture Notes in Computer Science **2498**, pp. 194–209.

*Accepted for publication October 11, 2002.*

**Roy Want** *Intel Corporation, 2200 Mission College Boulevard, Santa Clara, California 95054 (electronic mail: roy.want@intel.com).* Dr. Want is a principal engineer at Intel Research. His interests include ubiquitous computing, wireless protocols, hardware design, embedded systems, distributed systems, automatic identification, and micro-electromechanical systems (MEMS). He received his B.A. degree in computer science from Churchill College, Cambridge University, UK, in 1983 and continued research at Cambridge into reliable distributed multimedia systems. He earned a Ph.D. degree in 1988 and then went to Olivetti Research (1988–1991). While there, he developed the Active Badge, a system for automatically locating people in a building. Dr. Want joined the Ubiquitous Computing program at Xerox PARC in 1991 and led a project called PARCTab, one of the first context-aware computer systems. At PARC, he managed the Embedded Systems group and earned the position of principal scientist.

**Trevor Pering** *Intel Corporation, 2200 Mission College Boulevard, Santa Clara, California 95054 (electronic mail: trevor.pering@intel.com).* Dr. Pering is a research scientist at Intel Research. His research interests include many aspects of mobile and ubiquitous computing, including usage models, power management, novel form factors, and software infrastructure. He received his Ph.D. in electrical engineering from the University of California, Berkeley, with a focus on operating system power management techniques. Outside of engineering, he enjoys music (he plays jazz trombone), backpacking, and travel. He is a member of the ACM.

**David Tennenhouse** *Intel Corporation, 2200 Mission College Boulevard, Santa Clara, California 95054 (electronic mail: david.tennenhouse@intel.com).* Dr. Tennenhouse is an Intel Vice President and Director of Research. He has been one of the pioneers of asynchronous transfer mode (ATM) networking, active networks, software radio, and desktop media processing. He previously served as Chief Scientist and Director of the Information Technology Office of the Defense Advanced Research Projects Agency (DARPA), where he formulated the PRO-Active Computing research strategy of DARPA, which emphasizes the networking of embedded and autonomous systems. Dr. Tennenhouse received his B.A.Sc. and M.A.Sc. degrees from the University of Toronto. In 1989, he completed his Ph.D. at the Com-

puter Laboratory of the University of Cambridge, UK. He then joined the Massachusetts Institute of Technology, where he held appointments in the Department of Electrical Engineering and Computer Science and in the Sloan School of Management. He is a founder of a consulting firm with expertise in fault tolerant transaction processing and has been a consultant to a number of organizations. Dr. Tennenhouse is a member of the ACM and IEEE and served on the Visiting Committee on Advanced Technology of the National Institute of Standards and Technology. He has been a member of the Subcommittee on Computing Information and Communications R&D of the National Science and Technology Council and chaired the Technology and Policy Working Group of the President's Information Infrastructure Task Force.

# Dynamic reconfiguration: Basic building blocks for autonomic computing on IBM pSeries servers

by J. Jann
L. M. Browning
R. S. Burugula

A logical partition in an IBM pSeries™ symmetric multiprocessor (SMP) system is a subset of the hardware of the SMP that can host an operating system (OS) instance. Dynamic reconfiguration (DR) on these logically partitioned servers enables the movement of hardware resources (such as processors, memory, and I/O slots) from one logical partition to another without requiring reboots. This capability also enables an autonomic agent to monitor usage of the partitions and automatically move hardware resources to a needy OS instance nondisruptively. Today, as SMPs and nonuniform memory access (NUMA) systems become larger and larger, the ability to run several instances of an operating system(s) on a given hardware system, so that each OS instance plus its subsystems scale or perform well, has the advantage of an optimal aggregate performance, which can translate into cost savings for customers. Though static partitioning provides a solution to this overall performance optimization problem, DR enables an improved solution by providing the capability to dynamically move hardware resources to a needy OS instance in a timely fashion to match workload demands. Hence, DR capabilities serve as key building blocks for workload managers to provide self-optimizing and self-configuring features. Besides dynamic resource balancing, DR also enables Dynamic Capacity Upgrade on Demand, and self-healing features such as Dynamic CPU Sparing, a winning solution for users in this age of rapid growth in Web servers on the Internet.

One of the cardinal features of an autonomic component in an information technology (IT) infrastructure is the ability of the component to adapt itself smoothly to changes in its environment. Endowing a computing system with this self-management feature often translates to the implementation of self-protecting, self-healing, self-optimizing, and self-configuring algorithms and subcomponents. Because the primary role of an operating system (OS) is to manage the physical resources of a computer system so as to optimize the performance of its applications (including middleware, which consists of applications from the perspective of the OS), an OS supporting autonomic computing[1] needs to handle the changes in the amount of physical resources allocated to it in a smooth fashion. Some of the most prominent physical resources of an OS are processors, physical memory, and I/O devices.

The current tendency among the noncommodity symmetric multiprocessor (SMP) system vendors is to develop systems that are increasingly large in terms of the number of processors, number of I/O slots, and memory size. Although advances in the design of hardware continue to provide rapid increases in the

sizes of these physical resources, a number of major applications and subsystems often lag behind in scalability; hence, the trend in high-end SMPs is to support partitioning of large SMPs and to use these systems for effective server consolidation. Partitioned SMPs typically come in two kinds: systems with physical partitions (PPARs) and systems with logical partitions (LPARs). In a physically partitioned system, the granularity of partitioning is typically coarse, because the partitioning occurs at physical boundaries such as system boards. In a logically partitioned system, the granularity of partitioning is typically much more fine-grained, such as a single CPU or even a fraction of a CPU, a small block of memory, or an I/O-slot instead of an entire I/O-bus. Hence a given SMP can be subdivided into many more LPARs than PPARs.

IBM first provided LPAR support in the Advanced Interactive Executive (AIX*) operating system with the introduction of the pSeries* 690 system in December 2001. This first release of LPAR support was static in nature, that is, the reassignment of a resource from one LPAR to another LPAR cannot be made while AIX is actively running, and both the donor LPAR and the receiver LPAR must be rebooted to enable a reassignment. For such a system to provide support for various resource-related autonomic computing features, such as dynamic resource balancing across LPARs, Capacity on Demand, Dynamic CPU Sparing, and hot swapping, it needs to augment the static partitioning capabilities with dynamic LPAR (DLPAR) capabilities. As of 2002, the pSeries 690 supports the dynamic reassignment of resources across LPARs running AIX. In AIX, this functionality is referred to as dynamic reconfiguration (DR). Since AIX is an enterprise UNIX** operating system that has been designed to be robust, high in performance, rich in functions and support of platforms, and hence monolithic, the addition of a valuable autonomic computing feature such as DR has to be carefully morphed into the existing semantics, code base, and structural organization of the operating system. These challenges found in adding autonomic computing capabilities are encountered by most of the large systems with a significant installation base. Later in this paper, we briefly describe the design of DR in AIX and show that with carefully developed designs, autonomic computing capabilities can be added to an enterprise quality OS, while preserving its performance, semantics, and structural organization. Besides describing the designs within AIX that enable the smooth migration of physical resources, we also describe how these designs are being exploited to provide a variety of valuable autonomic computing features to an IT establishment.

**Autonomic benefits of DLPAR.** DLPAR in a pSeries 690-AIX system offers a great deal of flexibility to users, allowing resources to be shifted to where they are most needed without impacting system availability. The DLPAR technologies that have been developed provide the basic building blocks on which many self-optimizing, self-configuring, self-protecting, and self-healing features of the system are built. These features enable the implementation of autonomic system management and goal-oriented policies to optimize the performance and usage of system resources. DR also improves the levels of resource utilization and the reliability and serviceability (RAS) characteristics of the SMP, that translate into real cost savings for the IT establishment.

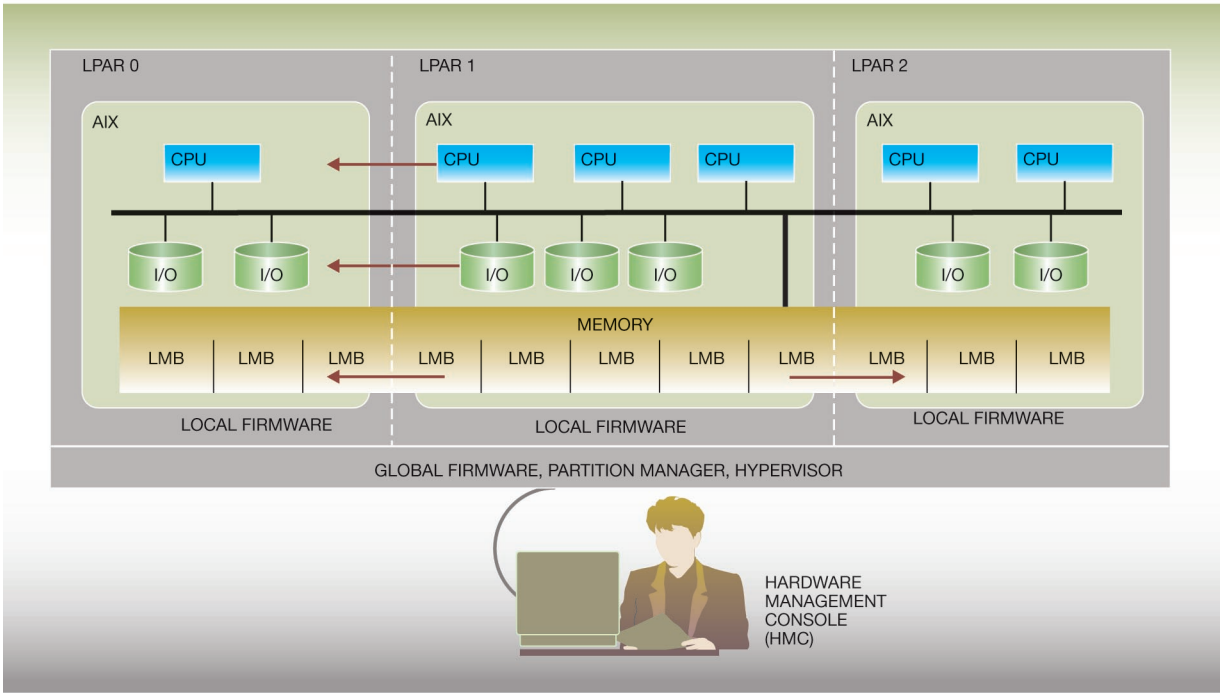
Some of the benefits offered by an SMP with LPAR capabilities are:

1. Servers can be consolidated by simply placing the workloads of several smaller servers into separate LPARs of a big SMP, hence reducing and unifying systems administration tasks.
2. Workloads can be separated by designating separate LPARs to run different workloads, for example, one LPAR for development work, one LPAR for testing, and several LPARs for production workloads.
3. The running of an application/subsystem/OS at its optimal performance and scalability can be obtained on an LPAR with optimal amounts of physical resources for that specific instance of application/subsystem/OS.

DLPAR additionally enables the following autonomic features in a system:

- Dynamic Capacity on Demand (DCOD)—DLPAR enables cross-partition workload management, which is particularly important for server consolidation, in that it can be used to better leverage system resources across partitions, thereby achieving higher levels of resource utilization, resulting in enhanced system throughput. Here is a possible usage scenario: The LPARs on an SMP are the servers for workloads originating from users in different time zones of the country, or even from different cities around the globe. While one LPAR "sleeps," its spare resources can be shifted to another LPAR that "wakes up" to do its work for the day. This shifting can be done manually via oper-

Figure 1 Logical partitioning in pSeries SMPs




ator command, and then later can be automated via the Global Resource Manager (GRM, an automated resource balancer across a specified group of LPARs in an SMP, based on OS utilization and needs) or the enterprise WorkLoad Manager (eWLM, an end-to-end response-time-based load balancer for "instrumented" applications spanning LPARs in an SMP, or even across SMPs).

- Dynamic Capacity Upgrade on Demand (DCUoD)—DLPAR enables the upcoming DCUoD feature of the pSeries 690 by allowing customers to purchase a server with extra unlicensed resource capacity, and later license and add this capacity dynamically to running AIX LPARs as their resource requirements increase.
- Dynamic CPU Guard and Dynamic CPU Sparing—DLPAR allows systems to smoothly replace processors that show intermittent, but correctable, errors. This self-healing feature will continue to become important with reduction in the silicon device size along with greater and greater integration on a chip. The Dynamic CPU Guard feature is an improved and dynamic version of the existing RAS feature named CPU Guard in earlier AIX versions. The older CPU Guard feature predicts the failure of a running CPU by monitoring certain types of transient errors and dynamically takes the CPU off line, but it does not provide a substitute CPU, so that a customer is left with less computing power. Additionally, the older feature will not allow an SMP to operate with less than two processors. The DLPAR technologies allow the OS to function even with one processor. In addition, the Dynamic CPU Sparing feature allows the transparent substitution of a good unlicensed processor for one that is suspected of being defective. This on-line switch is made seamlessly, so that applications and kernel extensions are not impacted. The new processor autonomously replaces the defective one. Dynamic CPU Guard and Dynamic CPU Sparing work together to protect a customer's investments through their self-diagnosing and self-healing software. Both features are planned to be available on pSeries 690 servers in AIX 5.2.

## The IBM pSeries DLPAR system architecture

The initial release of DR will be supported on the POWER4 pSeries 690 and 670 servers. Figure 1 illus-

trates the RS/6000* system architecture for DLPAR. In the diagram, LMB stands for logical memory block and is the granularity of physical and logical memory assigned to an LPAR. In AIX 5.2, an LMB will consist of 256 MB of contiguous memory. The size of an LMB is expected to decrease in future releases of AIX.

In this section we list some of the pSeries system components that had to be modified to become DR-aware in order to implement DLPAR. Some of these components were introduced during the implementation of static LPAR (e.g., the hardware management console, hypervisor, global firmware, and two registers for partition memory management), and some components existed even before LPAR existed (e.g., local firmware and AIX). We did not have to introduce any new components for the implementation of DLPAR; we only made changes to existing ones.

**Hardware management console.** The hardware management console (HMC) is the main control point for DLPAR configuration definitions and operations. In the initial release, these operations are controlled mainly through a new resource management graphical user interface (GUI) that runs on the HMC. This GUI invokes a new HMC command that encapsulates the DLPAR request and provides the necessary sequencing, so that the firmware and the OS can act in a coordinated fashion to achieve the desired result. Internally, this new HMC command notifies both the OS and the global firmware of a remove or add request. In time, this command will be used by other programs such as the GRM to dynamically transfer resources based on capacity requirements.

**Global and local firmware.** Global firmware provides the basic DLPAR enablement through machine-dependent logic that is designed to start, stop, and electronically isolate physical resources within the context of a logical partition. That is, global firmware performs these operations indirectly at the request of the OS, which actually passes these requests to the local firmware of the LPAR. In general, the local firmware does not contain machine-dependent logic, but is used to provide an abstraction to the OS of the logical resources that are currently assigned to the LPAR. The OS utilizes the local firmware interfaces to determine the identity and physical characteristics of logical resources that are present in the LPAR and to control them. For example, local firmware provides interfaces to start, stop, isolate, and unisolate logical resources by invoking functions provided by the global firmware.

**AIX.** When the OS receives a request to add or remove a resource, it has to take actions, both in the user space and in the kernel, to achieve the desired result. These actions are described in more detail in the next section.

## AIX DR components

Dynamic reconfiguration support in AIX 5.2 is provided for three types of hardware resources: processors, memory, and I/O slots. In this AIX release, the granularity of addition or removal for processors is one CPU; for memory, 256 MB; and for I/O, one PCI (Peripheral Component Interconnect) slot. In future releases of AIX, finer granularities are planned for both memory and processor. The following subsections briefly describe the design for the addition and removal of each hardware resource type, and the kernel modifications required for enabling DR in AIX. Although we mostly focus on DR as an enabler of autonomic computing in this paper, we have also used autonomic computing principles even within the DR design, an example of which is given in a subsequent subsection entitled "Dynamic removal of memory."

**DR of processors.** Achieving dynamic reconfiguration of processors introduced changes to the base kernel as follows:

1. For consecutive CPU identifiers (IDs), the kernel keeps track of its CPUs by assigning a distinct number, called a logical CPU ID, to each of its CPUs. Prior to DR, the kernel assumed that these logical CPU IDs were consecutive numbers, that is, no holes were allowed. To be able to randomly remove and add CPUs without perceptible disruption to applications, we had to modify the kernel so that it can tolerate missing items in the numbering of logical CPU IDs. Though we could have modified the kernel whichever way we wanted, there are many third-party applications (e.g., device drivers and performance tools) that assume consecutive numbering of CPUs, and they must be provided with binary compatibility, so as to fulfill the backward compatibility objective of AIX. Hence, another layer of numbering, bind CPU IDs, was introduced. The bind CPU ID abstraction provides a consecutive numbering of on-line CPUs for applications, even when the logical CPU IDs in the kernel are randomly removed and added, that is, the list of logical CPU IDs now becomes a list of on-line CPUs and off-line or removed CPUs.

2. For MP/UP locks, some components of the kernel were coded to decide at boot time whether they will acquire uniprocessor (UP) locks or multiprocessor (MP)-capable locks during the lifetime of the OS session. These components were modified so that they will function properly even when DR changes the system dynamically from a UP to an MP (and vice versa).

*Dynamic removal of processors.* Dynamic removal of a processor involves the following tasks initiated from the OS:

1. Notify DR-registered applications and kernel extensions, if any, so that they will voluntarily remove dependencies on the CPU to be removed. This task typically involves unbinding the threads that are bound to the CPU being removed.
2. Migrate threads bound to this CPU to another running CPU.
3. Retarget pending interrupts, and change the bindings of all interrupts currently bound to the processor to be removed. This task involves changing the interrupt controller data structures.
4. Migrate the timers and threads from the CPU being removed to another CPU within the same LPAR.
5. Notify the hypervisor or firmware to complete the removal task.

*Dynamic addition of processors.* Dynamic addition of a processor involves the following tasks initiated from the OS:

1. Create a process (waitproc) for idle looping on the incoming CPU before it starts to do real work.
2. Set up various hardware registers (e.g., SDR1 = Search Descriptor Register 1, which defines the start physical address and size of the page table in memory, GPR1 (General Purpose Register 1) for kernel stack, GPR2 for the kernel table of contents, etc.) of the incoming CPU.
3. Allocate or initialize, or both, the processor-specific kernel data structures (dispatcher run-queue, per-processor data area, interrupt stack, etc.) for the incoming CPU.
4. Add support for the incoming CPU to the interrupt subsystem.
5. Notify the DR-registered applications and kernel extensions that a new CPU has been added.

**DR of memory.** The implementation of dynamic reconfiguration of memory in AIX 5.2 enables the removal and addition of 256 MB contiguous sections of memory. This unit is referred to as a logical memory block (LMB). A smaller-sized LMB, for example, 16 MB, will be allowed in future releases of AIX.

Two important challenges were resolved during the implementation of memory DR: In the first one, the physical addresses of some memory are exposed to manipulation by applications over which the kernel does not have direct control. These accesses are allowed for functional reasons in some cases (e.g., direct memory access, or DMA), and for performance reasons in other cases (e.g., pretranslated addresses). Pretranslated addressing is an internal feature of AIX with which virtual-to-physical address translations for a data buffer to be involved in a DMA operation is done only once for the life of the data buffer. DR of page-frames with these uses can be handled in at least two ways: (1) by modifying the kernel and firmware so that such accesses by kernel extensions to the page-frame being removed can be controlled; or (2) by requiring kernel subsystems to register a DR callback function with the kernel DR subsystem, and invoking the callback function at memory removal time.

We have applied both techniques, choosing one over the other, depending on the specific circumstances, while minimizing the impact on system performance as well as kernel changes. In the case of DMA, the kernel and firmware control the access to the memory being removed by selectively disabling the bus traffic while the contents of the memory with DMA are being migrated to a new location. In the case of pretranslated addresses, a callback mechanism is used.

The second challenge is enabling the translation-on execution of the majority of the kernel, which previously was run in translation-off mode and hence required maximally sized data structures to be allocated at boot time for the maximum amount of physical memory that the AIX/LPAR instance can potentially grow into.

*Dynamic removal of memory.* For the purpose of dynamic removal, we classify the memory page frames of AIX into five categories: unused, pageable, pinned, DMA-mapped, and translation-off memory. The approach taken to remove a page-frame (4096 bytes) in each of these categories is as follows:

1. A page-frame containing a free page is simply removed from its free-list.

2. A page-frame containing a pageable page can be made to either page out its contents to disk or to migrate its contents to a different free page-frame. In the future, an autonomous agent, for example, GRM or eWLM, can choose one of these two approaches, depending on its knowledge of memory availability.

3. A page-frame containing a pinned page will have its contents migrated to a different page-frame; also the page-fault reload handler had to be made to spin during the migration.

4. A page-frame containing a DMA-mapped page cannot be removed or have its contents migrated until all the accesses to the page are blocked. Here, the term "DMA-mapped page-frame" is used generically to mean a page-frame whose physical address is subject to read or write by an external (to kernel) entity such as a DMA engine. The contents of a DMA-mapped page-frame are migrated to a different page-frame with a new hypervisor call (*h_migrate_dma*) that will selectively suspend the bus traffic while it is modifying the TCEs (translation control entries) in system memory used by the bus unit controller for DMA accesses.

   Other page-frames whose physical addresses are exposed to external entities are handled by invoking preregistered DR callback routines and then waiting for completion of the removal of their dependencies on the page.

5. A page-frame containing translation-off pages will not be removed by DR in AIX 5.2. Fortunately, there is just a small amount of these page-frames, and they are usually colocated in low memory. These page frames are intended to be handled in later AIX releases.

The design and implementation of dynamic memory removal has manifested itself as three modular functions, such that one can mix and match these functions in several possible ways, adapting to the state of the system at the time of memory removal, thus achieving the desired end result with the most optimal path. This design adheres to the self-optimizing principles of autonomic computing, as described in Reference 2. These three modular functions perform the following three tasks respectively on the memory (LMB) being removed: (a) remove its free and clean pages from the regular use of VMM (Virtual Memory Manager), (b) page-out its pageable dirty pages, and (c) migrate the contents of each remaining page-frame in the LMB to a free page-frame outside the LMB. Memory removal can be implemented with any one of the sequences: abc, ac, bc, or just c.

For example, the decision to either invoke page-out (task b) or to migrate (task c) all the pages depends on the load in the LPAR at that particular time. If the LMB being removed contains a lot of dirty pages that belong to highly active threads, then it does not make sense to invoke task b, because these pages will be paged back in almost immediately, negatively impacting the efficiency of the system.

As a second example, if there are not enough free frames in other LMBs to migrate the pages to, then the memory removal procedure can invoke task b before invoking task c, so that there will be far fewer pages left that need to be migrated in task c.

***Dynamic addition of memory.*** When a memory-add request arrives at AIX, it has to perform two tasks: allocate and initialize software page descriptors that will hold meta-data for the incoming memory, and distribute the incoming memory among several free-frame pools so as to preserve the behavior of memory management algorithms. The challenges encountered in implementing these two tasks and how they were resolved are now described.

The primary challenge was the allocation of memory for the software page descriptors for the incoming memory. The problem was that, prior to DR, these page descriptors could be accessed in translation-off mode while trying to reload a page mapping into the hardware page table. If the page descriptors are allowed to be accessed in translation-off mode, the memory allocated for those new descriptors has to be physically contiguous with the memory for existing descriptors, which implies that memory has to be reserved at boot time for descriptors for the maximal amount of memory that the OS instance can potentially grow into. This can potentially incur inefficiency and wastage of much memory, particularly if not utilized. We avoided this wastage by changing the kernel so that software page descriptor data structures are always accessed in translation-on mode.

Another challenge that was resolved while implementing dynamic memory addition was the difficulty in distributing the incoming memory across different page replacement daemons, so that each dae-

mon handles a roughly equal load. In AIX, memory is hierarchically represented by the data structures vmpool, mempool, and frameset. A vmpool represents an affinity domain of memory. A vmpool is divided into multiple mempools, each mempool being managed by a single page replacement least recently used (LRU) daemon. Each mempool is further subdivided into one or more framesets that contain the free-frame lists, so as to improve the scalability of free-frame allocators. When new memory is added to AIX, the vmpool that it should belong to is defined by the physical placement of the memory chip. Within that vmpool, we want to distribute the memory across all the available mempools to balance the load on page replacement daemons. However, the kernel assumed that a mempool consisted of physically contiguous memory. Thus, to be able to break up the new memory (LMB) into several parts and distribute them across different mempools, the kernel was modified to allow mempools to be made up of discontiguous sections of memory.

**DR of I/O slots.** The methods to dynamically configure or unconfigure a device have been introduced as early as AIX version 3. The changes required in the kernel design for DR of I/O slots were not in the same scale as those for DR of processors, and particularly those for DR of memory. The reason is that the onus of configuring or unconfiguring a device lies with the device driver software, which operates in the kernel extension environment. The kernel just acts as a provider of serialization mechanisms for devices accessing common resources and as an intermediary between the applications and the device drivers.

## Challenges of adding autonomic features to a mature UNIX OS

The AIX kernel is an industrial strength pageable and pre-emptable kernel that offers high performance and scalability (up to 32-way SMP) and supports many vendor device drivers, databases, and applications. The same kernel source code supports all reasonable combinations of 32-bit or 64-bit kernels, 32- or 64-bit applications, and 32- or 64-bit RISC (reduced instruction-set computer) systems with old and new RISC architectures, uniprocessors, and multiprocessors. Being friendly to its users, AIX goes out of its way to offer backward binary compatibility in a wide variety of situations. Being robust and having high performance, the critical kernel sections are skillfully guarded by a carefully crafted hierarchy of data and code locks. Implementing DR involved careful changes to numerous critical components of the AIX kernel.

## Self-healing and self-protecting features

DR also serves as the foundation for a new advanced self-healing and self-protecting technology, especially when it is coupled with the presence of extra unlicensed capacity. This new technology enhances a preexisting self-diagnosing technology, called the CPU Guard feature, that monitors recoverable error rates for processors. At its simplest level, this new technology substitutes a spare processor in a transparent fashion for a processor that the system has internally diagnosed as being defective. Spare resources are only present if the system was shipped with extra unlicensed capacity as defined by the Capacity Upgrade on Demand solution, although it should be noted that there are no license keys that have to be entered to enable this new Dynamic CPU Sparing technology. This Dynamic CPU Sparing feature does not address the case in which a CPU fails so suddenly that a state-save and an orderly live swap of the CPU cannot be performed.

The technical aspects of this new technology are outlined next. Firmware monitors the health of each processor in terms of the number of recoverable errors. If this number exceeds an internal threshold, it raises a repeat guard error to the operating system and makes available an unlicensed processor, assuming that one is available. AIX acts on this error notification by invoking the DR Manager, which guides the self-protecting procedure from the perspective of an operating system. If an unlicensed processor is not available, it proceeds to take the defective processor off line in a fashion similar to that of the existing CPU Guard feature; this procedure constitutes a self-protecting feature of the system. If an unlicensed processor is available, the procedure uses it as a substitute for the defective one, making the switch transparently with the new DR technologies. This action constitutes a self-healing feature of the system.

The DR Manager determines whether an unlicensed processor is available through the use of new firmware routines, and if it finds one, attempts to take ownership of it from the firmware by invoking new firmware routines. These new firmware routines are the same ones that are used for DR processor addition, although a different return code is used to indicate that they are reserved for self-healing. Next, the DR Manager queries the kernel to determine the

identity of the defective processor, which was named by the repeat guard error log entry. Finally, it invokes the kernel to perform the live swap.

The new processor is always started before the defective one is stopped, so that this technology can be applied to a single-processor LPAR—an improvement over the old CPU Guard self-protecting technology, which could not remove the last two on-line processors. At a high level, the switch is made by taking over the execution of the defective processor and using it to control the sequence of events that are required to transparently complete the switch, much of which has to be run on the defective processor itself, since the new processor is going to assume the logical identity of the failing one eventually. This is largely a matter of preserving the physical state of the defective processor and atomically reprogramming any funneled hardware interrupts that may be directed to the defective physical processor before allowing the new processor to begin operating.

The kernel actually performs the switch by masking external interrupts on the defective processor, so that it is not expected to respond to external events that may be generated by adapters or other processors in the partition, and by tightly controlling the execution of the new processor. Before starting the new processor, the defective processor saves the state of its registers, so that they can be restored by the new processor once it is functioning. Next, it vacates its logical CPU ID inside the kernel so that the new processor can be brought up in the proper logical CPU ID relative to the kernel. This is important in that logical workloads assigned to the defective logical processor (e.g., its threads and timers) do not need to be migrated, which in large part constitutes the transparency of the switch. There are many logical processor states and very few physical processor states that need to be taken care of. Next, it invokes firmware to start the new processor at a startup routine that is specific to this algorithm, and it waits for the new processor to indicate that it has successfully added itself to the global processor interrupt queue. Once this occurs, the defective processor can stop itself without fear of losing any external interrupts. This synchronization point is required to ensure that the global processor interrupt queue has at least one processor at all times. At this point, the new processor loads the register state that was previously preserved and resumes the execution path previously established by the defective processor.

This self-healing technology is built into the AIX base operating system and is automatically enabled by default, although the customer may disable it through system management options. This new technology is initially available on the pSeries model 690.

### Self-optimizing and self-configuring features

The next release of AIX will have provisions for the automatic movement of its logical resources between LPARs, allowing the overall utilization of the physical resources of an SMP to be increased at no extra cost to the installation. For example, an agent, such as a Global Resource Manager (GRM), will monitor the utilizations, needs, and Service Level Agreements (SLAs) of a pool of LPARs and autonomously move the DR resources from an underutilized LPAR to a needy one, hence enriching the SMP with self-optimizing and self-configuring capabilities in a timely fashion. GRM also ensures that the DR movements will be orchestrated in a smooth fashion and without undesirable oscillations.

Additionally, as part of the IBM autonomic initiative, work is ongoing to provide optimal end-to-end response time for "instrumented" Web and commercial applications that span the LPARs of an SMP, as well as applications that span SMPs with or without LPARs, based on SLAs.

## Conclusion

The DLPAR and DR technologies that have been developed on the IBM pSeries 690 servers have enabled these servers to become truly autonomic computer servers. As described in this paper, these servers have features that are self-protecting and self-healing, and they have the basic building blocks that enable these systems to be self-configuring and self-optimizing. The self-configuring and self-optimizing features are currently planned to be available in the near future. Thus, all four self-managing features that are at the heart of an autonomic server will soon be available on the pSeries 690. DLPAR is a strategic pSeries AIX capability and will also be made available on future pSeries servers.

In a possible future enhancement, virtualization of a physical processor into virtual processors allows the processor resource to be sharable in a fine-grained fashion (instead of one processor at a time) among a pool of LPARs in the SMP.

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of The Open Group.

## Cited references


1. P. Horn, *Autonomic Computing: IBM's Perspective on the State of Information Technology*, IBM Corporation (October 15, 2001); available at http://www.research.ibm.com/autonomic/manifesto/autonomic_computing.pdf.
2. K. Ekanadham et al., *Anatomy of Autonomic Server Components*, Research Report RC 22637, IBM T. J. Watson Research Center, Yorktown Heights, NY.



*Accepted for publication August 16, 2002.*



**Joefon Jann** *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (electronic mail: joefon@us.ibm.com).*

Ms. Jann is a Senior Technical Staff Member at the Watson Research Center, where she leads a small team that conceived and prototyped the notion of DLPAR for AIX. She is currently working on automating DLPAR. Her previous projects in IBM include the design and prototype of a DSM (distributed shared memory) system for the pSeries servers, the design and implementation of the LoadLeveler® hierarchical communication infrastructure, which enables IBM's LoadLeveler product to work in the largest SP™ installations. She coinvented the "Jann MPP Workload Model," was a member of the Deep Blue Computer chess team, a developer of the IBM product VMPRF (VM Performance Reporting Facility), a VM/SNA Area Specialist, and an APL programmer. Ms. Jann was a lecturer in mathematics at Lehman College for three years, and holds B.A. and M.A. degrees in pure mathematics from Wellesley College–MIT and the City University of New York (CUNY), respectively, and an M.S. degree in computer science from Columbia University. She is a member of the IEEE.


**Luke M. Browning** *IBM Server Group, 11501 Burnet Road, Austin, Texas 78758 (electronic mail: browninl@us.ibm.com).*

Mr. Browning is a Senior Technical Staff Member in the AIX Kernel Architecture and Design Department of the IBM Server Group, working on dynamic LPAR, workload management, threads, and process management. He joined IBM in 1984 in Austin after receiving his bachelor of science degree in computer science from the University of Texas at Austin.


**R. Sarma Burugula** *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (electronic mail: burugula@us.ibm.com).*

Mr. Burugula is an advisory software engineer at the Watson Research Center. He received his B.S. degree from Regional Engineering College Warangal and an M.S. degree from the Indian Institute of Technology Kanpur in India, both in computer science. He joined IBM in 1996 and has been working primarily on the IBM pSeries platform, developing various parallel and scalable subsystems.

**A technical view of autonomic computing**
October 2002

Tivoli software

# How Tivoli software products support the IBM Autonomic Computing Initiative™

*Tivoli software: Intelligent management software that integrates and automates*

Contents



## Introduction

The high-tech industry has spent decades creating systems of ever-increasing complexity to solve a wide variety of business problems. Today complexity itself has become part of the problem. After deployment, hardware and software problems occur, people make mistakes and networks grow and change. Improvements and changes in performance and capacity of IT components can require constant human intervention. A machine waiting for a human to tune it and fix it can translate into lost dollars.

With the expense challenges that many companies face, IT managers want to improve the return on investment of IT by reducing the total cost of ownership, improving the quality of service and managing IT complexity.

Autonomic computing helps address these issues and more by using technology to manage technology. Autonomics is a term derived from human biology. In the same way that your body's autonomic system monitors your heartbeat, checks your blood-sugar level and keeps your body temperature at 98.6° F without any conscious effort on your part, autonomic computing components anticipate needs and resolve problems—without human intervention.

IBM products with autonomic capabilities can deliver customer value with their predictive and proactive functions that anticipate changing conditions and problems. This paper defines the customer value of autonomic computing, the requirements for achieving an autonomic environment, the steps for successful implementation and the products that are making this computing concept a reality.

## The customer value of autonomic computing

Autonomic computing was conceived of as a way to help reduce the cost and complexity of owning and operating the IT infrastructure. In an autonomic environment, IT infrastructure components—from desktop computers to mainframes—are self-configuring, self-healing, self-optimizing and self-protecting. These attributes are the core values of autonomic computing.


*Self-configuring*
With the ability to dynamically configure itself on the fly, an IT infrastructure can adapt—with minimal human intervention—to the deployment of new components or changes in the IT environment.

*Self-healing*
A self-healing IT infrastructure can detect when IT components fail and can cure or work around those component failures to provide continued availability of business applications.

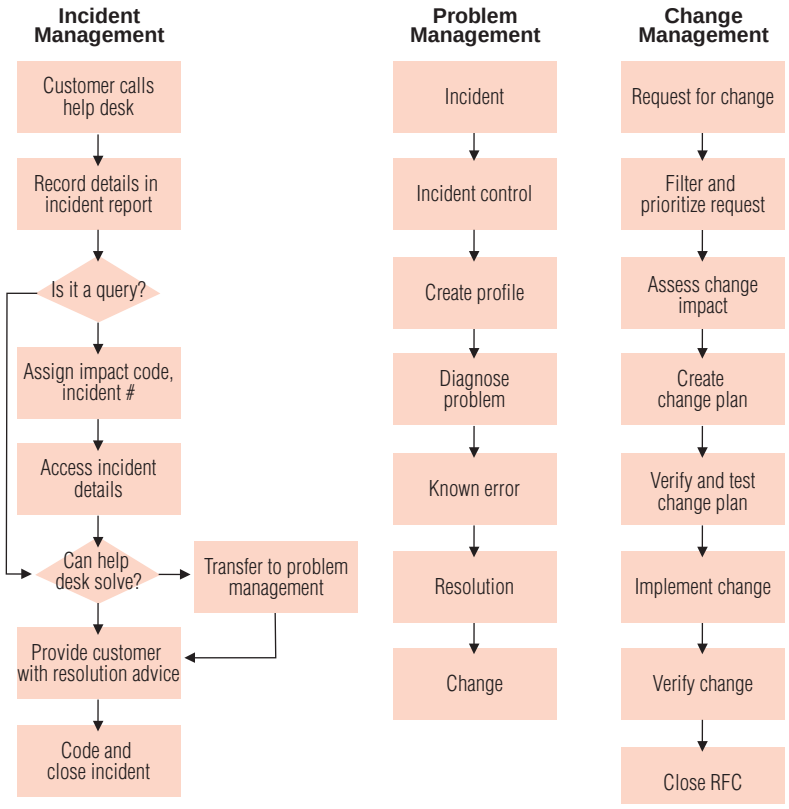
*Self-optimizing*
Self-optimization is the ability of the IT environment to efficiently address resource allocation and utilization with minimal human intervention.

*Self-protecting*
A self-protecting IT environment can detect hostile or intrusive behavior as it occurs and take autonomous actions to make itself less vulnerable to unauthorized access and use, viruses, denial-of-service attacks and general failures.

In an autonomic environment, components work together and communicate with each other and with high-level management tools. They regulate themselves and, sometimes, each other. They can proactively manage the network while hiding the inherent complexity of these activities from end users.

The IBM view of autonomic computing is to make its software behave automatically and bring the autonomic systems management capability to the infrastructure, enabling the IT environment—including systems management software—to configure, optimize, heal and protect itself.

Typically a complex IT infrastructure is managed using a set of IT management processes. Industry initiatives, including IT Infrastructure Library and IBM IT Process Model, define best practices for managing the IT environment. The diagram below shows an example of a typical process flow for incident management, problem management and change management. The actual mechanics of how these flows are implemented in a particular IT organization can vary, but the basic functionality is usually the same.




The efficiency and effectiveness of these processes are typically measured using metrics like elapsed time of a process, percentage executed correctly, skill requirements, average cost of execution and so on. Autonomic computing technology can help improve the efficiency and speed with which these processes can be implemented by automating some steps in the process.

*Quick process initiation:* Typical implementations of these processes require a human to initiate the process (create the request for change, collect incident details, open a problem record). This usually requires the IT professional

to spend time gathering the right information. In a self-managing system, components can initiate the processes based on information derived directly from the system. This helps reduce the manual labor and time required to respond to critical events.

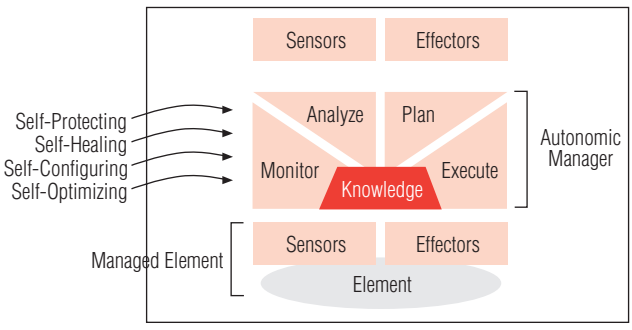
*Reduced time and skill requirements:* Tasks or activities in these processes usually stand out as skills-intensive, long-lasting and difficult to complete correctly the first time because of system complexity. In a change management process such an activity is change impact analysis, and in problem management such an activity is problem diagnosis. In self-managing systems, resources are instrumented so that the expertise required to perform these tasks can be encoded into the system, helping reduce the amount of time and skills needed to perform these tedious tasks.

The self-managing capability of the IT environment helps improve responsiveness, reduce total cost of ownership and improve time to value. It can help reduce the total cost of ownership because the IT professional can complete the IT processes at a low average cost, and it can help accelerate time to value because it reduces the time it takes to execute an IT process.

The remaining sections discuss the autonomic computing technology and tools that help make it possible.

### Autonomic computing architecture concepts

The architecture shown in the diagram below identifies the required architectural elements in an autonomic environment. The architecture is organized into two major elements—a managed element and an autonomic manager.




*Structure of self-management technologies*

The managed element is the resource being managed. At this level of the architecture, the element targeted by management could be a single resource or a collection of resources. The management element exports sensors and effectors. Sensors provide mechanisms to collect information about the state and state transition of an element. Effectors are mechanisms that change the state of an element.

Sensors and effectors represent the instrumentation interface that is available to an autonomic manager. The autonomic manager is a component that implements the control loop. The architecture decomposes the loop into four parts:

- ***Monitor***—*Mechanisms that collect, aggregate, filter, manage and report details (metrics, topologies and so on) collected from an element.*
- ***Analyze***—*Mechanisms to correlate and model complex situations (time series forecasting, queuing models). These mechanisms allow the autonomic manager to learn about the IT environment and help predict future situations.*
- ***Plan***—*Mechanisms to structure the action needed to achieve goals and objectives. The planning mechanism uses policy information to guide its work.*
- ***Execute***—*Mechanisms that control the execution of a plan with considerations for on-the-fly updates.*

The monitor, analyze, plan and execute parts of the autonomic manager relate to the functionality of most IT processes. For example, the mechanics and details of IT processes like change management and problem management are different, but it is possible to abstract these into four common functions—collect the details, analyze the details, create a plan of action and execute the plan. These four functions correspond to the monitor, analyze, plan and execute components of the architecture.

The analyze and plan mechanisms are the essence of an autonomic computing system, because they encode the know-how to help reduce the skill and time requirements of the IT professional.

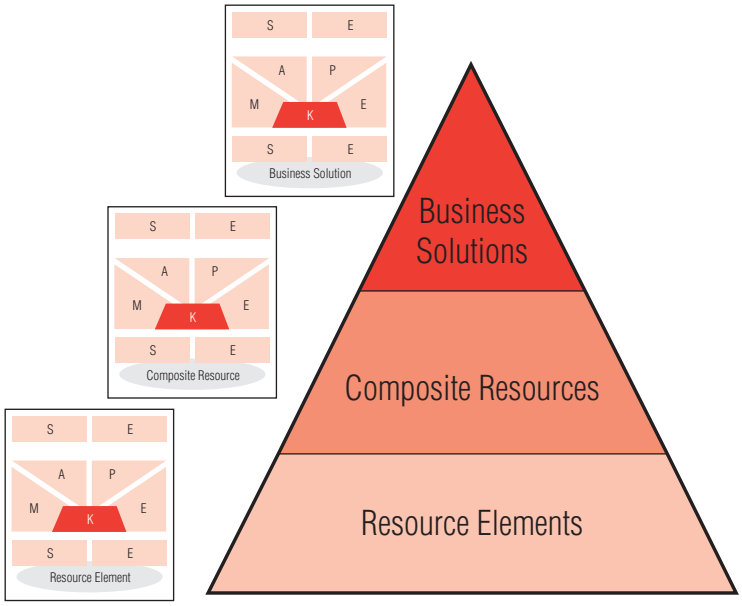
The knowledge part of the autonomic manager is where data and information used by the four components of the autonomic manager is stored and shared. Knowledge that can be found here includes policy, topology information, system logs and performance metrics.

The architecture prescribes a second set of sensors and effectors. This second set enables collaboration between autonomic managers. Autonomic managers can communicate with each other in a peer-to-peer context and with high-level managers.

Each autonomic self-management attribute of self-configuring, self-healing, self-optimizing and self-protecting is the implementation of the intelligent control loop (in an autonomic manager) for different operational aspects of configuration, healing, optimization and protection. For example, an autonomic manager can self-configure the system with the correct software if software is missing. By observing a failed element, it can self-heal the system by restarting it. It can self-optimize the current workload if increased capacity is observed. If an intrusion attempt is detected, it can self-protect the systems by blocking the intrusion at the perimeter and by verifying the resource.

### Autonomic computing in the IT environment

To understand how autonomic computing plays a role in different parts of the IT environment, it is important to view the IT environment at different levels. Self-management within each level involves implementing control loops to allow individual resources, composite resources and business solutions to monitor, analyze, plan and execute changes to their environment.



IBM provides a suite of management products that helps enable automation of routine management tasks for individual resource elements. IBM products, including the IBM® Tivoli® Monitoring family, IBM Tivoli Configuration Manager, IBM Tivoli Access Manager and IBM Tivoli Storage Manager, begin to bring self-managing capabilities to the IT infrastructure for resource elements (systems, applications, middleware, networks and storage devices). IBM is working through IBM Server Group, IBM Software Group and a variety of third parties to embed the appropriate technologies and enable resource elements to participate in the autonomic IT infrastructure.

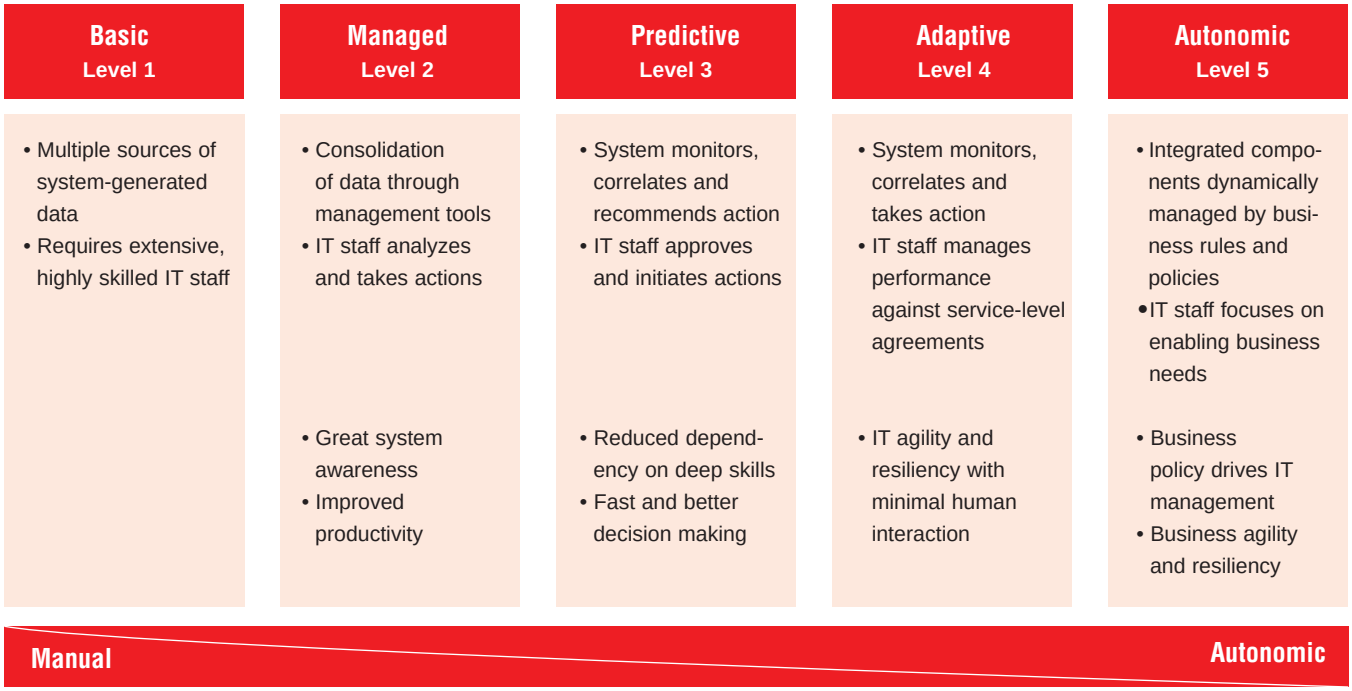
At the composite resource level, the evolution to autonomic computing is enabled by the evolution to transaction-based management. In the past, resource elements were traditionally grouped by type (all servers), by location (all servers within a department or facility) or by function (all Web servers). As enterprises develop e-business environments, resources are increasingly aggregated within a transactional context spanning heterogeneous resources. For example, servers, applications, databases and storage devices that touch e-business transactions would be grouped separately from those assigned to human resources. If the composite resource grouping is homogenous (such as a server cluster) or heterogeneous (such as a Web server, database and storage system), the performance and availability requirements of different transaction types drive the autonomic activity on individual resource elements. The attainment of service-level objectives for IT transactions causes resources to be dynamically assigned, configured, optimized and protected for changing business workloads. IBM Tivoli Monitoring for Transaction Performance, IBM Tivoli Storage Resource Manager, IBM Tivoli Identity Director and Tivoli Configuration Manager are examples of IBM products that work together to enable the evolution to autonomics at the composite resource layer.

The highest layer of the IT environment is a business solution, such as a customer care system or an electronic auction system. The business solution layer requires autonomic systems management solutions that comprehend the state of business processes—based on policies, schedules, trends and service-level objectives and their consequences—and drive the appropriate behavior for transactional systems and their underlying individual resources. Business-aware IBM products include IBM Tivoli Service Level Advisor, IBM Tivoli Business Systems Manager and IBM Tivoli Systems Automation for S/390®.

### Autonomic computing levels

Making the IT infrastructure autonomic is an evolutionary process enabled by technology, but it is ultimately implemented by each enterprise through the adoption of these technologies and supporting processes.

The diagram below represents how an IT environment evolves towards a truly autonomic environment, from basic through managed, predictive, adaptive and finally to a fully autonomic e-business environment.




1. The basic level represents a starting point where some IT environments are today. Each infrastructure element is managed independently by IT professionals who set it up, monitor it and eventually replace it.

2. At the managed level systems management technologies can be used to collect information from disparate systems onto fewer consoles, helping reduce the time it takes for the administrator to collect and synthesize information as the IT environment becomes more complex.

3. At the predictive level new technologies are introduced to provide correlation among several infrastructure elements. These elements can begin to recognize patterns, predict the optimal configuration and provide advice on what course of action the administrator should take.

4. As these technologies improve and as people become more comfortable with the advice and predictive power of these systems, they can progress to the adaptive level. The IT environment can automatically take actions based on the available information and the knowledge of what is happening in the environment.

5. To get to the fully autonomic level the IT infrastructure operation is governed by business policies and objectives. Users interact with autonomic technology tools to monitor business processes, alter | objectives, or both.

The following sections discuss the autonomic computing levels for each autonomic characteristic—self-configuring, self-healing, self-optimizing and self-protecting. This can help you determine your current level of readiness, assess the capabilities of current tools and evaluate it within the context of a longer-term view.

#### Self-configuring

An enterprise can greatly increase its responsiveness to both employees and customers with a self-configuring IT environment. With the ability to dynamically configure itself on the fly, an IT infrastructure can adapt immediately—and with minimal human intervention—to the deployment of new components or changes in the IT environment. For example, an e-business retailer dealing with seasonal workload peaks during the holiday shopping season or increased business for a particular event can use a self-configuring IT infrastructure to reassign servers from underutilized pools to overutilized ones to match shifting workloads. Tivoli software management tools from IBM can allow you to provision a wide range of resources, including systems, applications, users and access privileges, and physical and logical storage. Monitoring and event correlation tools can help determine when changes in the IT infrastructure warrant reconfiguration actions. These tools can allow you to reconfigure your IT environment within minutes or hours rather than in days or weeks.

IBM has defined five implementation levels for a self-configuring IT infrastructure, based on the major capabilities that should ultimately exist for true autonomic functionality.

| Basic<br>Level 1 | Managed<br>Level 2 | Predictive<br>Level 3 | Adaptive<br>Level 4 | Autonomic<br>Level 5 |
|---|---|---|---|---|
| **Definition**<br>• Reliance on system reports, product documentation and manual actions to configure, optimize, fix and protect individual IT components<br>• Human intervention required to perform any action | **Definition**<br>• Software provides monitoring, consolidation, facilitation and automation of IT tasks<br>• Clusters may be used to balance workload<br>• Analysis and actions performed by IT staff | **Definition**<br>• Individual IT components and system tools monitor and analyze changes<br>• User and role-based reallocation of resources<br>• Capacity is adjusted at the resource level to help provide transactional performance | **Definition**<br>• IT components individually and collectively monitor, analyze and take action with minimal human intervention<br>• IT policy determines capacity reallocation across specific resource types | **Definition**<br>• IT components collectively and automatically managed by business rules and policies established in the system<br>• Resources discovered and capacity reallocated |
| **Value**<br>Single systems configured and administered | **Value**<br>Helps increase administrator productivity through fast system installation | **Value**<br>Helps improve administrative productivity through role-based provisioning | **Value**<br>Helps increase system availability and reduces human intervention through event-driven automated provisioning | **Value**<br>Helps achieve business resource optimization to meet service-level objectives and business priorities |

*Level 1: Basic*

The focus is on the ability to deploy, configure and change an individual system component, including system hardware configuration, storage hardware configuration, communication configuration and operating system configuration. Basic resource-specific tools are used to perform configuration actions. Configuration of multiple resources is done by logging on to each resource admin tool separately to perform the configuration action.

*Level 2: Managed*

The focus is on the ability to deploy and manage change to an aggregated group of systems. This includes managing multiple systems and system images, moving systems in and out of clusters, deploying applications to

groups of machines, managing groups of users and moving storage in and out of storage networks. The concept of virtualized storage is introduced and the ability to monitor the collective system health and storage components to make decisions about how they might need to be reallocated and reconfigured is assessed.

*Level 3: Predictive*
The notion of managing based on role is introduced, including user role and system role, so that configurations can be appropriately tailored for their use. Configuration sensing (for example, inventory scanning) and runtime monitoring information is used to determine when corrective actions need to be taken. The administrator can initiate corrective actions based on system recommendation.

*Level 4: Adaptive*
The focus is on dynamically managing the configuration of the environment by leveraging sophisticated correlation and automation. The key notion is that the reconfiguration happens automatically and the IT infrastructure adjusts itself based on overall configuration health and role changes.

*Level 5: Autonomic*
Reconfiguration actions are taken within the context of overall business policies and priorities. Business impacts are assessed to determine the appropriate reconfiguration actions. It also includes the ability to provision proactively and anticipate issues that might jeopardize service levels before breaches actually occur.

#### Self-configuring capabilities enabled by Tivoli software products

Tivoli software products that can be used to implement a self-configuring environment include:

*Tivoli Configuration Manager*
Tivoli Configuration Manager configures automatically to rapidly changing environments. It provides an inventory scanning engine and

a state management engine that can sense when software on a target machine is out-of-synch with a reference model for that class of machine. It can automatically create a customized deployment plan for each target and sequence the installation of software in the right order.

*Tivoli Identity Manager*
Tivoli Identity Manager automates user lifecycle management and integrates with HR and native repositories. It uses automated role-based provisioning for account creation. The provisioning system communicates directly with access-control systems to help create accounts, supply user information and passwords and define account entitlements.

*Tivoli Storage Manager*
Tivoli Storage Manager provides self-configuring capabilities to perform tasks such as automatically identifying and loading the appropriate drivers for the storage devices connected to the server. Configuration and policy information can be defined once at a Tivoli Storage Manager configuration server and then propagated to a number of managed Tivoli Storage Manager servers. Policies and internal automation allow automatic extension of the server database, recovery log, or both when administrator-defined thresholds are reached.

### Self-healing

A self-healing IT infrastructure can detect improper operation of systems, transactions and business processes (either predictively or reactively) and then initiate corrective action without disrupting users. Corrective action could mean that a component is altered or other components are altered to accept its workload. Day-to-day operations do not falter or fail because of events at the component level. The Tivoli software availability management portfolio from IBM provides tools to help customers monitor the health and performance of their IT infrastructure. These tools help allow monitoring of multiple metrics from a heterogeneous collection of resources and provide the ability to perform filtering, correlation and analysis. Based on the analysis, automated actions can be taken to cure problems even before they occur. Autonomic

capabilities are provided at multiple levels to allow customers to understand business impacts and proactively manage the availability of the IT infrastructure. Workbench tools allow integration of third-party applications.

IBM has defined five implementation levels for self-healing and availability management, based on the major capabilities that should ultimately exist for true autonomic functionality.

| Basic<br>Level 1 | Managed<br>Level 2 | Predictive<br>Level 3 | Adaptive<br>Level 4 | Autonomic<br>Level 5 |
|---|---|---|---|---|
| **Definition**<br>• Local consoles on each system used for administration<br>• Manual reading of logs is the primary method for problem debug<br>• IT availability reports are created manually | **Definition**<br>• Centrally monitor and report on IT resources<br>• Event filtering, correlation and management products used centrally<br>• Problems fixed manually by skilled administrators | **Definition**<br>• IT resource contribution to transactional delays monitored and projected<br>• Business impact (IT services impact) of resource problems understood<br>• Helps automate corrective actions for routine problems | **Definition**<br>• Policy-based, desired-state driven to return infrastructure to functioning state<br>• Problem trends identified and automatically corrected | **Definition**<br>• Dynamic reprovisioning of IT infrastructure to accommodate resource failures<br>• Business continuity automatically maintained |
| **Value**<br>Helps minimize capital expenditures in the IT cost center | **Value**<br>Helps increase ROI of IT resources and IT services | **Value**<br>IT resources managed in context of business impact | **Value**<br>Helps increase business availability | **Value**<br>Helps maximize business impact of IT services |

*Level 1: Basic*
Systems administration and problem management is accomplished by using significant human processing power. Availability of systems is addressed in a reactive way. IT staff learns of problems from customers complaining about lack of service. Problem determination, correlation and cures are accomplished with a great deal of human intervention. Highly skilled IT staff are needed to debug problems.

*Level 2: Managed*
The focus is on the ability to collect and view availability information from remote locations. Many resources may be located outside the data center, perhaps in branch offices. Error logs can be accessed remotely. IT has deployed a set of monitoring tools to report on availability to a central location. Multiple system and network events can be filtered or manually correlated to identify the root cause of problems. Problems are fixed by skilled administrators.

*Level 3: Predictive*
IT has granular views into IT systems to accurately pinpoint the cause of outages. Complex, multiple metric collection is now possible, instead of single metrics. Filtering is now advanced and tied to correlation engines, allowing improved root-cause problem determination to take place. Automated corrective actions are taken to known problems. These capabilities help customers prioritize which problem to repair first, based on the business impact of the outage.

*Level 4: Adaptive*
Systems can automatically discover, diagnose and fix problems on multiple monitored resources (operating system, application, middleware) across multiple monitored systems. The IT infrastructure availability is maintained automatically to keep in tune with predefined desired states. Outages don't bring down the system; it dynamically adapts to outages until repairs can be made to maintain service levels. For example, thresholds are temporarily raised to account for added workload.

*Level 5: Autonomic*
Problem determination and diagnosis depend on sophisticated knowledge already encoded in the system about components and their relationships. The inference capability allows the system to automatically figure out corrective actions within the right business context. For example, if a particular outage cannot be contained with available resources, lower-priority business applications may be shut down or run with degraded quality of service to keep higher-priority business applications functioning.

**Self-healing capabilities enabled by Tivoli software products**

Tivoli software products that can be used to implement a self-healing environment include:

*IBM Tivoli Enterprise Console*

Tivoli Enterprise Console® collates error reports, derives root cause and initiates corrective actions. The event server and correlation engine help allow cross-resource correlation of events observed from hardware, applications and network devices throughout an enterprise. Events from multiple resources can be analyzed in realtime to automatically highlight the critical problems that merit attention versus the misleading symptoms and effects. After a problem is highlighted, the system takes self-healing actions by responding automatically when possible or efficiently guiding the support staff to the appropriate response.

*IBM Tivoli Switch Analyzer*

Tivoli Switch Analyzer correlates network device errors to the root cause without user intervention. It is a Layer 2 switch network management solution that provides automated Layer 2 discovery. It identifies the relationship between devices, including Layer 2 and Layer 3 devices, and identifies the root cause of a problem without human intervention. During a network event storm it can filter out extraneous events to correlate the true cause of the problem.

*IBM Tivoli NetView*

Tivoli NetView® helps enable self-healing by discovering TCP/IP networks, displaying network topologies, correlating and managing events and SNMP traps, monitoring network health and gathering performance data. Router fault isolation technology quickly identifies and focuses on the root cause of a network error and initiates corrective actions.

*Tivoli Business Systems Manager*

Tivoli Business Systems Manager collects realtime operating data from distributed application components and resources across the enterprise and provides a comprehensive view of the IT infrastructure components that make up different business solutions. It contains technologies that analyze how an outage would affect a line of business, critical business process or service-level agreement (SLA).

*Tivoli Systems Automation S/390*
Tivoli Systems Automation S/390 manages realtime problems in the context of an enterprise's business priorities. It provides monitoring and management of critical system resources, such as processors, subsystems, Sysplex Timer and coupling facilities. It supports self-healing by providing mechanisms to reconfigure a processor's partitions, perform power-on reset on IML processors and IPL operating systems (even automatically), investigate and respond to I/O configuration errors, and restart and stop applications if failures occur.

*IBM Tivoli Risk Manager*
Tivoli Risk Manager enables self-healing by assessing potential security threats and automating responses, such as server reconfiguration, security patch deployment and account revocation. This helps enable system administrators who are not security experts to monitor and assess security risks in realtime with a high degree of integrity and confidence across an organization's multiple security checkpoints. This product contains technology from IBM Research.

*IBM Tivoli Monitoring for Applications, IBM Tivoli Monitoring for Databases and IBM Tivoli Monitoring for Middleware*
This family of products minimizes vulnerability by discovering, diagnosing and reacting to disruptions automatically. It provides monitoring solutions and a local automation capability through a set of Proactive Analysis Components. A sophisticated resource model engine allows for local filtering of monitored data, raising events when specific conditions are met. Local rules can be encoded to take immediate corrective action, providing automatic recovery for server failures.

*Tivoli Storage Resource Manager*
Tivoli Storage Resource Manager automatically identifies potential problems and executes policy-based actions to help prevent or resolve storage issues, minimize storage costs and provide application availability. It can scan and discover storage resources in the IT environment. It supports policy-based automation for the allocation of storage quotas and storage space, monitors file systems and provides reports on capacity and storage asset utilization.

### Self-optimizing

Self-optimization is the ability of the IT infrastructure to efficiently maximize resource allocation and utilization to provide Quality of Service for both system users and their customers. In the near term self-optimization primarily addresses the complexity of managing system performance. In the long term self-optimizing software applications may learn from experience and proactively tune themselves in an overall business objective context. Workload management uses self-optimizing technology to help optimize hardware and software use and verify that service-level goals are being met. Predictive analysis tools provide views into performance trends, allowing proactive action to be taken to help optimize the IT infrastructure before critical thresholds are exceeded.

IBM has defined five implementation levels for a self-optimizing IT infrastructure that can optimize workloads and transaction performance across multiple resources.

| Basic<br>Level 1 | Managed<br>Level 2 | Predictive<br>Level 3 | Adaptive<br>Level 4 | Autonomic<br>Level 5 |
|---|---|---|---|---|
| **Definition**<br>• Monitor and report on individual IT resource performance in disparate formats<br>• Manually optimize the performance of individual IT resources | **Definition**<br>• Monitor and report on end-user performance (transactions)<br>• Centralized, comprehensive performance visibility<br>• Manually optimize groups of linked resources | **Definition**<br>• Service-level agreements and priorities are committed and managed<br>• Realtime service-level visibility and projected future-level visibility<br>• Analysis and recommendations for manual tuning | **Definition**<br>• Policy-based, desired-state driven<br>• Service-level priorities drive automatic resource (re)configuration and shift workload to meet objectives<br>• Automated closed-loop resource tuning | **Definition**<br>• Dynamic reprovisioning of aspects of IT infrastructure to meet shifting workloads and service-level commitments<br>• Automated closed-loop transaction tuning |
| **Value**<br>Can minimize capital expenditures in the IT cost center | **Value**<br>Helps increase ROI of IT resources and IT services | **Value**<br>Predictable performance and value of IT services | **Value**<br>Flexibility, fast deployment of new IT services | **Value**<br>Helps maximize business impact of IT services |

*Level 1: Basic*

Individual resources provide point data regarding individual component performance or utilization, allowing users a simple view of how workload affects a single system. Basic tools allow dynamic viewing of components,

but comprehensive views of system performance are still put together manually by looking at multiple local views and reports available with the resource-specific tools.

*Level 2: Managed*
Management tools allow information on resource utilization and performance to be gathered and collected in a central location. Simple, comprehensive transaction views are possible using techniques such as roundtrip measurements, synthetic transactions or end-user client-capture capabilities. Many resources in the middle of a transaction are invisible or not instrumented, and many resources are often located outside the data center—perhaps in branch offices or other locations. Optimizing the IT components is still done manually and with trial and error.

*Level 3: Predictive*
Tools now provide value by creating detailed, comprehensive transaction views and can break down the composite view of the transaction across the resource elements. Resources can be grouped by transaction types, service levels can be monitored and automated tools provide notifications of impending violations—allowing manual reconfiguration of the IT environment. Predictive tools can perform trend analysis on historical data and provide recommendations.

*Level 4: Adaptive*
Instrumentation is now available on the composite resources to allow changes of status and automated balancing of work when overload or underload conditions exist across resources in the environment. This level of control provides users with the ability to manage comprehensive performance and effectively meet SLAs.

*Level 5: Autonomic*
Workload balancing and transaction optimization is done within the business context. Business trade-offs are expressed in machine-processable format, allowing IT management tools to dynamically reallocate resources based on varying business needs. Automated tuning of servers, storage and networks takes place to maintain quality of service for high-priority business applications.

**Self-optimizing capabilities enabled by Tivoli software products**

Tivoli software products that can be used to implement a self-optimizing environment include:

*Tivoli Service Level Advisor*

Tivoli Service Level Advisor helps prevent SLA breaches with predictive capabilities. It performs trend analysis based on historical performance data from Tivoli Enterprise™ Data Warehouse and can predict when critical thresholds could be exceeded in the future. By sending an event to Tivoli Enterprise Console, self-optimizing actions can be taken to help prevent the problem from occurring.

*IBM Tivoli Workload Scheduler for Applications*

Tivoli Workload Scheduler for Applications automates, monitors and controls the flow of work through the IT infrastructure on both local and remote systems. It can automate, plan and control the processing of these workloads within the context of business policies. It uses sophisticated algorithms to maximize throughput and help optimize resource usage.

*Tivoli Business Systems Manager*

Tivoli Business Systems Manager enables optimization of IT problem repairs based on business impact of outages. It collects realtime operating data from distributed application components and resources across the enterprise and provides a comprehensive view of the IT infrastructure components that make up different business solutions. It works with Tivoli Enterprise Console to enable self-optimizing actions to help prevent poor performance from affecting a line of business, critical business process or SLA.

*Tivoli Storage Manager*

Tivoli Storage Manager supports Adaptive Differencing technology to help optimize resource usage for backup. With Adaptive Differencing, the backup-archive client dynamically determines an efficient approach for creating backup copies of just the changed bytes, changed blocks or changed files, delivering improved backup performance over dial-up connections. These technologies allow just the minimum amount of data to be moved to backup, helping optimize network bandwidth, tape usage and management overhead.

*Tivoli Monitoring for Transaction Performance*
Tivoli Monitoring for Transaction Performance helps customers tune their IT environments to meet predefined service-level objectives. It enables organizations to monitor the performance and availability of their e-business and enterprise transactions to provide a positive customer experience. It integrates with the Tivoli Enterprise Console environment for alerting and proactive management, helping enable optimization of resource usage from a transactional perspective.

*IBM Tivoli Analyzer for Lotus Domino*
Tivoli Analyzer for Lotus® Domino™ contains a Proactive Analysis Component that allows administrators to verify the availability and optimal performance of Lotus Domino servers. It provides intelligent server health monitoring and expert recommendations to correct problems.

### Self-protecting

A self-protecting IT environment can take appropriate actions automatically to make itself less vulnerable to attacks on its runtime infrastructure and business data. These attacks can take the form of unauthorized access and use, malicious viruses that can format hard drives and destroy business data, and denial-of-service attacks that can cripple critical business applications. A combination of security management tools and storage management tools are necessary to deal with these threats. Security management tools can help businesses consistently enforce security and privacy policies, help reduce overall security administration costs and help increase employee productivity and customer satisfaction. Critical configuration changes and access-control changes should only occur with the right approvals. Tools should detect violations of security policy, and if necessary, automated actions should be taken to minimize risk to IT assets. Tivoli software storage management tools help enable businesses to automatically and efficiently back up and protect business data. Autonomic security and storage solutions provide administrators with a way to create policy definitions and express event correlation and automation knowledge.

IBM has defined five implementation levels for a self-protecting IT infrastructure.

| Basic<br>Level 1 | Managed<br>Level 2 | Predictive<br>Level 3 | Adaptive<br>Level 4 | Autonomic<br>Level 5 |
|---|---|---|---|---|
| **Definition**<br>• Security controls are implemented on each device (individual formats)<br>• Basic infrastructure tools are deployed to protect corporate assets<br><br>**Value**<br>Can minimize capital expenditures in the IT cost center | **Definition**<br>• Security configurations for devices are managed centrally<br>• Intrusion sensor devices are deployed throughout the infrastructure<br><br>**Value**<br>Helps improve ability to protect corporate assets | **Definition**<br>• Application security is managed according to common security policy<br>• Correlation of intrusion events to identify real threats from normal business activity<br>• Consolidated sign-on<br><br>**Value**<br>Helps improve application security with a common security infrastructure | **Definition**<br>• Security access is granted to users and enforced dynamically based on user-policy changes<br>• Infrastructure intrusion-detection tool configuration adapts to IT security policy and threats<br><br>**Value**<br>Security infrastructure responds with minimal human intervention | **Definition**<br>• e-business security context<br>• Self-collaborating systems for detection, verification and reconfiguration<br>• Learned policies for future protection<br><br>**Value**<br>Helps integrate perimeter and application security to business policy |

*Level 1: Basic*
Localized security configuration requires administrators to configure each component independently and manually track changes. Local backup and recovery tools are used to protect data. Audit reports are on a per-machine basis. A great deal of human intervention is required to protect the runtime and business data.

*Level 2: Managed*
Management tools are used to centralize security administration, allowing the centralized creation of user IDs and controlling access privileges to resources. Intrusion sensing and auditing tools are used to collect data about intrusion attempts. These are manually reviewed, and corrective actions are taken to protect against future attacks. Centralized backup and recovery tools provide an incremental backup capability across multiple resources.

*Level 3: Predictive*
Security policies can be consistently administered across the enterprise manually, using enterprisewide security management tools. IDs and access privileges are coordinated across multiple applications and can be consistently

revoked if necessary. Perimeter sensors can detect security violations and correlate them to detect attacks. Security tools provide recommendations for corrective action.

*Level 4: Adaptive*
Security management focuses on advanced automation, including automatically enabling new users and disabling IDs for those who leave. It automatically grants access to systems and applications needed to do a new job while disabling access to systems associated with the old job. If access-control violations and intrusions are detected, automatic reconfiguration actions are initiated to help quarantine systems and disable access to IDs.

*Level 5: Autonomic*
The focus is on learning systems and systems that can adapt lower-level resource policies in response to higher-level business policy. Collaboration across system components makes it possible to reconfigure systems on the fly, automatically apply security patches when necessary, modify intrusion monitoring levels based on business needs and adapt policies to help prevent future problems based on past history.

**Self-protecting capabilities enabled by Tivoli software products**
Tivoli software products that can be used to implement a self-protecting environment include:

*Tivoli Storage Manager*
Tivoli Storage Manager self-protects by automating backup and archival of enterprise data across heterogeneous storage environments. Scaling to protect thousands of computers running a dozen operating system platforms, its intelligent data movement and store techniques and comprehensive automation help reduce administration costs and increase service levels.

*Tivoli Access Manager*
The Tivoli Access Manager family of products self-protects by helping prevent unauthorized access and using a single security policy server to enforce security across multiple file types, applications, devices, operating systems and protocols. It supports a broad range of user authentication methods, including Web single sign-on, and has the ability to control access to many types of resources for authenticated users.

*Tivoli Identity Manager*

Tivoli Identity Manager self-protects by centralizing identity management, integrating automated workflow with business processes and leveraging self-service interfaces to increase productivity.

*Tivoli Risk Manager*

Tivoli Risk Manager provides systemwide self-protection by assessing potential security threats and automating responses, such as server reconfiguration, security patch deployment and account revocation. It collects security information from firewalls, intrusion detectors, vulnerability scanning tools and other security checkpoints. It simplifies and correlates the vast number of events and alerts generated by numerous security point products and quickly identifies the real security threats to help administrators respond with adaptive security measures.

*IBM Tivoli Privacy Manager for e-business*

Tivoli Privacy Manager for e-business self-protects by automating many privacy-compliance activities, simplifying the incorporation, monitoring and enforcement of privacy policy into business processes. It can record end user's opt-in and opt-out choices according to the policy, can be used to monitor and enforce access according to the privacy policy and can create audit trail reports.

## Summary

Companies want and need to reduce their IT costs, simplify management of their IT resources, realize a fast return on their IT investment and provide high levels of availability, performance, security and asset utilization. Autonomic computing helps address these issues. IBM is a leader in the evolution to autonomic computing and offers integrated systems management solutions for resource management, transaction-oriented management and business-solution management that span the four autonomic computing disciplines of self-configuring, self-healing, self-optimizing and self-protecting.

## To learn more

For information on Tivoli software and integrated solutions from IBM, contact your IBM sales representative or visit **ibm.com/tivoli**



IBM Corporation
Software Group
Route 100
Somers, NY 10589
U.S.A.

Printed in the United States of America
10-02


The Tivoli home page on the Internet can be found at **ibm.com/tivoli**

The IBM home page on the Internet can be found at **ibm.com**



e business software

G000-0000-00

**IBM Global Services**

*Executive Tek Report*

***March 22, 2002***

# Autonomic computing: Can it help to manage the increasingly complex information environment?

Executive Summary - Information systems that respond to changing conditions and regulate and repair themselves are the dream behind autonomic computing. This initiative depends on bringing together advanced technologies and management techniques in a holistic way. Autonomic computing isn't just an academic "grand challenge." It is an answer to a looming crisis in our ability to manage an ever more complex information environment, where the costs and scarcity of trained personnel threaten to halt progress. Autonomic computing can help ease this increasing complexity by moving it into systems and using system resources to manage the problem. To be successful, autonomic computing depends on pioneer work in architectural design, use of standards, agent software, context detection and much more.

*Your firm has just signed a contract to provide retirement investment support for a global chemical company. There are four key elements in this contract: a portfolio of investment options, employee education, legal qualification and automated payroll services. Your portfolio is long established through partnerships with brokers worldwide. The biggest change will be a dramatic increase in the number of transactions. That can be handled by the FastFarm server farm, which will allocate additional resources, as needed. You'll venture into new business areas with employee education, subcontracting with a specialist to provide both simulation and just-in-time (JIT) education. The JIT systems will interface directly with your client's normal knowledge management system so employees will get advice on an investment bond as easily as they get information on how to store chemicals in the lab. To fulfill your commitment on legal qualification, the application must run transparently, using both profiles of employees and links to government systems to ensure that restrictions, estimations on taxes, and other obligations are accurate and localized. A key feature is the generation of alerts for employees when new regulations make a significant difference to their current investments. Finally, the automated payroll services in this contract allow for employee initiated changes to be handled directly. While the software "handshakes" do not present a challenge, this element has a high level of security controls and requires bonded experts to safeguard the combined operation.*

**How this happens.** Today, to troubleshoot an application installation, get different platforms to work together or manage storage resources, we usually rely on human intervention. The complexity created by increasing computing power, higher bandwidth communications, greater connectivity and a proliferation of devices is handled by highly trained experts who diagnose, design and build solutions. This is enormously expensive, ever more challenging and, ultimately, a losing game, as the number of people required to manage the system grows and grows. At current rates, it is estimated that, globally, demand for skilled IT workers will to increase by over 100 percent in the next six years.

Putting the complexity into the system and letting *it* bear most of the burden of solving these problems is the idea behind autonomic computing. By taking advantage of new and emerging technologies and management approaches, the future may include systems that function well with limited intervention and that provide a simplified user experience. According to the paper, *Autonomic Computing: IBM's Perspective on the State of Information Technology*, this depends on at least eight elements.

*1. To be autonomic, a computing system needs to "know itself" and comprise components that also possess a system identity.* To effectively manage itself, a system must first have a full inventory of all the software, hardware and connections it "owns," including those that it can rent or borrow. It needs to have specific policies on claiming and relinquishing resources. In short, it has to know its own boundaries. This is the biggest set of tasks currently handled by humans in terms of contractual obligations, but tools can help. For instance, a software tool could be used to discover and inventory the applications and devices that are connected to the system. System management software (SMS) can provide realtime indications of the use of shared resources. Architecture tools can establish the roles of components and how they work together. Agents can constantly monitor the system to determine when new devices are linked, when applications are added or updated, and when users turn on new capabilities. In the future, sensors might be able to provide context by determining the real world environment.

*2. An autonomic computing system must configure and reconfigure itself under varying and unpredictable conditions.* Every time something is added or subtracted from a complex system, many elements of the system are affected. This is visible today when you install or uninstall a program, and the system asks what you want to do about updating or removing shared files. The dependencies and options multiply rapidly as systems become larger and more interconnected. Success here is highly dependent on sophisticated, flexible architectural designs, which may be explored and validated through simulation. Modular software, adapters, development and adherence to industry standards and use of Extensible Markup Language (XML) are also keys to success here. Many of the aspects of grid computing are directly applicable.

*3. An autonomic computing system never settles for the status quo -- it always looks for ways to optimize its workings.* Users need flexibility and dependability as their priorities shift. Business managers need to have confidence that they can make decisions based on the opportunities they face, rather than the limits of their IT environment. Systems should be as large as they need to be, and no bigger. Ultimately, optimization keeps costs of hardware and maintenance from spiraling out of control and improves the speed and efficiency of the system. Here, management tools that handle capacity and demand and can quickly respond to emerging problems are essential. Control theory and decision support can also help make this level of optimization a reality. The most significant results in optimization have come from deep computing, where processing power and algorithms have been put to work to generate and choose options for the best use of resources.

*4. An autonomic computing system must perform something akin to healing -- it must be able to recover from routine and extraordinary events that might cause some of its parts to malfunction.* Rebooting is not the ideal solution to a malfunctioning system. Communications systems already detect trouble spots and can route around them and activate other resources to maintain system reliability. Computing systems need to be able to do the same thing. Error checking software already exists to detect and even anticipate and correct emerging problems, but much still needs to be done to automate root cause analysis. Once problems are discovered, there is a need to be able to reallocate resources on the fly, take advantage of any redundancy, find capacity that can be rented or borrowed and identify the best options for the system as a whole. Sensors and agents may help here. Actual self-repair is likely in the future.

*5. A virtual world is no less dangerous than a physical one, so an autonomic computing system must be an expert in self-protection.* In today's connected world, the potential for system intrusion and virus attacks must be addressed. The threats are getting more sophisticated, spread more quickly and can cause greater damage. Some of the tools that are available to control access and anticipate and deal with malefactors are immune systems, pattern recognition, social network analysis, visualization, single sign-on and biometrics. In addition, open source approaches may help by leveraging a large, interested community to find and close security holes before they do harm. Ultimately, security features that are built into the system from the beginning provide the most hope.

*6. An autonomic computing system knows its environment and the context surrounding its activity and acts accordingly*. Two things are at work here: One context is the computing environment -- which specifies which resources might become available and the extent of sharing that is permissible. The other is the user's context -- which determines what information is of interest. For example, a global positioning system (GPS) might make it clear that a temperature query pertaining to the inside of a house located in the U.S. is best answered in Fahrenheit, and how it might be presented best (such as PDA, cell phone or workstation). Technologies including grid computing, sensors, user profiles, hard portals, workflow and decision support help provide contextual information and make the information received valuable.

*7. An autonomic computing system cannot exist in a hermetic environment.* This is a fundamental concept. Today, information systems depend on connectivity for most of their value. There is an ecosystem of relationships, and new devices and applications must arrive adapted to this ecosystem – not with a cluster of needs that must be met before they can fit into the ecosystem. Standards, nonproprietary software, open source software, grid computing adapters and agents all have roles to play here.

*8. Perhaps most critical for the user, an autonomic computing system will anticipate the optimized resources needed while keeping its complexity hidden.* Instead of *people* having to learn and adapt to computing systems, the *systems* should learn and adapt to us. At a minimum, the design should take advantage of what is known about humans so the system can be intuitive. Raising the bar on ease-of-use needs to take place at every level, from the casual user, all the way to an executive managing the information needs of dynamic partnering. Advanced interfaces, "anticiparallelism" (where the system performs, in a parallel fashion, anticipatory work to gauge what the user wants next), visualization and instinctual computing are just some of the technologies on the horizon to help achieve this.

**What this means to you.** Autonomic computing can assist in reducing market concerns over security, reliability, service level agreements and return on investment. Most of the focus in autonomic computing to date has been centered on technology. Given the seriousness of the challenges, this is appropriate today. However, for the vision to become a reality, focus must be shifted to user needs and to the services that will be required as the vision is realized. Interfaces need to adapt more to user experience and context. IT planning and strategy must become more future-focused and work beyond the impediments to autonomic computing progress. IT experts

will need to handle new levels of flexibility and an environment with very different security risks. Outsourcing models will change and disciplines -- such as IT architecture -- will need to be directed at solving problems for larger, more complex systems.

The importance of autonomic computing to the future of business is clear. Advances in autonomic computing can enable the evolution of computing to move forward at a rate faster than today, as IT people can leave more and more time consuming tasks for the system to complete, and they concentrate their efforts on new development. Much of the growth and many of the products and services envisioned for the future can only be realized if the benefits autonomic computing promises arrive.

**Two industries: Retail and Finance.** For the retail industry, autonomic computing may enable practical dynamic partnering. This would allow for rapid changes in relationships all along the supply chain, without losing data, so that operations can be optimized. In addition, as complexity is hidden, a richer array of information devices can be put to use to view, sample and purchase commodities. Both the contexts of transactions and the profiles of buyers have the potential to become more important components of the retail experience.

For financial services firms, autonomic computing may force more capability-based business designs by enabling rapid morphing of structure, without impact to the infrastructure and -- particularly -- its security features. As confidence grows, firms may be able to reach beyond known sources of data, creating new categories of information that can be put to use within risk parameters. This will provide broader, more real-world views and help enable better decision-making.

| Tek to watch |
| --- |
| Open source |
| XML |
| Encryption |
| Agents |
| Sensors |
| Grid computing |
| Simulation |
| Pervasive computing |

**References**

Horn, Paul. (October, 2001). Why autonomic computing will reshape IT. Retrieved from the World Wide Web, March 5, 2002. http://www.control.com/sqposting_html?pid=1003266873

Meehan, Michael. (November, 2001). Users Eye Self-Healing Systems Management Software. *Computerworld*. Retrieved from the World Wide Web, March 5, 2002. http://www.computerworld.com/itresources/rcstory/0,4167,KEY18_STO65313,00.html

Various. (November, 2001.) Autonomic Computing: A Grand Challenge. *EvoNet.* Retrieved from the World Wide Web, March 5, 2002. http://evonet.dcs.napier.ac.uk/evoweb/news_events/news_features/nf_article165.html

Various. Autonomic computing: IBM's perspective on the state of information technology. Retrieved from the World Wide Web, March 5, 2002. http://www.research.ibm.com/autonomic/manifesto/

**Other sites of interest**

Deep computing
http://www.research.ibm.com/dci/

Grid computing
http://www.ibm.com/services/insights/etr_grid.html

O'Reilly and Associates, Inc.
http://www.xml.com/

Single sign-on
http://www.ibm.com/services/insights/etr_singlesign-on.html

**About this publication**

*Executive Tek Report* is a monthly publication intended as a heads-up on emerging technologies and business ideas. All the technological initiatives covered in *Executive Tek Report* have been extensively analyzed using a proprietary IBM methodology. This involves not only rating the technologies based on their functions and maturity, but also doing quantitative analysis of the social, user and business factors that are just as important to its ultimate adoption. From these data, the timing and importance of emerging technologies are determined. Barriers to adoption and hidden value are often revealed, and what is learned is viewed within the context of five technical themes that are driving change:

**Knowledge Management**: capturing a company's collective expertise wherever it resides – in databases, paper, people's minds -- and distributing it to where it can produce the big payoffs
**Pervasive Computing**: combining communications technologies and an array of computing devices (including PDAs, laptops, pagers and servers) to allow users continual access to the data, communications and information services
**Realtime**: "a sense of ultracompressed time and foreshortened horizons, [a result of technology] compressing to zero the time it takes to get and use information, to learn, to make decisions, to initiate action, to deploy resources, to innovate" (Regis McKenna, *Real Time*, Harvard Business School Publishing, 1997.)
**Ease-of-Use**: using user-centric design to make the experience with IT intuitive, less painful and possibly fun
**Deep Computing**: uses unprecedented processing power, advanced software and sophisticated algorithms to solve complex problems and derive knowledge from vast amounts of data

This analysis is used to form the explanations, projections and discussions in each *Executive Tek Report* issue so that you not only find out *what* technologies are emerging, but *how* and *why* they'll make a difference to your business. If you would like to explore how IBM can help **you** take advantage of these new concepts and ideas, please contact us at **insights@us.ibm.com**. To browse through other resources for business executives, please visit: **ibm.com**/services/insights

*Executive Tek Report* is written by Peter Andrews, an IBM emerging technology analyst, and is published as a service of IBM Corporation.



G510-1659-00

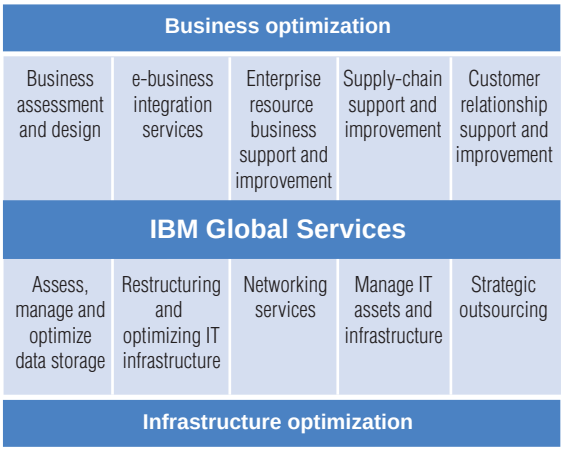
# IBM Global Services and autonomic computing

### Introduction

IBM Global Services has been an industry-leading IT services provider since it was formed in the mid-1990s. Providing its customers with technical and business consulting, outsourcing and learning services, its vast delivery capabilities and thought leadership help enterprises worldwide optimize their businesses and IT infrastructures.

IBM Services offerings recognize that an IT infrastructure comprises technology, process and organizational components, collectively referred to as the infrastructure "fabric." Technology components of the fabric (including such tangible elements as servers, software, storage and networks) are often acquired directly by customers for their own integration and deployment. IBM Global Services also might be a provider of these technology components through its many different outsourcing and outsourcing-related services offerings.

IBM Global Services assists customers today with process and organization components of the fabric by providing related consulting and design services such as Systems Management Consulting and Design; Systems Management Services; Infrastructure Resource Management Services; and Performance Management, Testing and Scalability Services. These services are described in more detail later in this paper.




Armed with the latest methodologies, benchmarks and intellectual capital, IBM consultants are already well-prepared to provide practical advice that addresses a customer's most pressing infrastructure concerns. As with technology components of the fabric, process and organizational assistance might be provided as an element of an outsourcing solution.

### Autonomic computing and IBM Global Services

Autonomic computing, an exciting new approach to systems management, can provide systems with the ability to perform management activities based on situations they observe or sense in the IT environment. This offers significant opportunities to increase the business value of a firm's IT infrastructure by reducing costs, increasing business flexibility and improving overall IT service quality. IBM Global Services supports the five levels of autonomic computing maturity as described in other autonomic computing materials and depicted on page 2. These levels relate to the attainment of an "ideal state" based on advanced, delivered and planned technologies and the latest IBM Global Services thinking derived from customer experiences, research and subject matter expertise. Autonomic computing defines a vision for the ideal state. Autonomic maturity levels are defined as basic, managed, predictive, adaptive and autonomic.

| Basic<br>Level 1 | Managed<br>Level 2 | Predictive<br>Level 3 | Adaptive<br>Level 4 | Autonomic<br>Level 5 |
|---|---|---|---|---|
| • Multiple sources of system-generated data<br>• Requires extensive, highly skilled IT staff | • Consolidation of data through management tools<br>• IT staff analyzes and takes actions | • System monitors, correlates and recommends actions<br>• IT staff approves and initiates actions | • System monitors, correlates and takes actions<br>• IT staff manages performance against SLAs | • Integrated components dynamically managed by business rules/policies<br>• IT staff focuses on enabling business needs |
| | • Greater system awareness<br>• Improved productivity | • Reduced dependency on deep skills<br>• Fast and better decision making | • IT agility and resiliency with minimal human interaction | • Business policy drives IT management<br>• Business agility and resiliency |

**Manual** **Autonomic**

New services that provide e-business on demand™, IT Optimization and Business and IT Resiliency were introduced recently, recognizing the increasing priority that business executives are placing on IT infrastructure and the new opportunities offered by autonomic computing. Another recent offering, e-business Management Services, is one of the IT industry's first services offering known to feature automated e-business processes and the ability to predict, identify and intercept business problems in realtime. This holistic approach to business continuity advances well beyond the insurance mentality of simple IT recovery. These and other offerings from IBM Global Services support the IBM Autonomic Computing Initiative™.

As infrastructure maturity progresses, process and organization maturity must evolve as well. For example, the self-configuring facet of autonomic computing can work properly only if there exists a knowledge of the configuration that is comprehensive in its content, at or near 100 percent accuracy and sufficiently current. This level of maturity in the inventory, configuration and asset processes as yet lies beyond today's best practices; the journey toward it should be taken in steps. IBM Infrastructure Resource Management Services already exist to extend the scope, reliability and usefulness of an organization's configuration information.

Similarly, autonomic computing allows for systems to "agree" on the rules by which neighboring systems can negotiate to share workload across resources. For this to occur—whether the resources are within an organization or across cooperating enterprises—requires organizational constructs that allow such dynamic partnering. In this scheme, decisions are required in-flight, with the governance scheme providing enablement. IBM consultants already are tackling issues such as this for clients in examining how e-business solutions can be set up through extended supply chains.

**Current IBM Global Services offerings**

IBM Global Services offerings are in place to assist customers in their path toward autonomic computing. These offerings are described in detail below.

IBM **Systems Management Consulting and Design** helps customers overcome the challenges of managing and operating enterprise information systems. IBM professionals combine experience with innovative thinking to help customers define their long-term IT systems management strategy, design administrative processes and select tools to manage systems, and create an organizational structure that enables continuous evaluation of systems performance against dynamic business needs. This includes Strategy and Assessment, Framework Design, Detail Design and Development and Deliver Services.

IBM **Systems Management Services** help deliver a flexible, scalable, reliable and secure infrastructure, responding to today's ever-changing environments without draining your resources and negatively affecting your business. IBM can provide the people, processes and technology to help you rapidly deploy the infrastructure and systems management capabilities that allow you to maximize return on IT investments and grow your business.

IBM Global Services professionals partner with clients to integrate new and existing systems across organizational and geographic boundaries. From legacy platforms to e-business infrastructures to pervasive computing, IBM can provide skills transfer while helping you design and implement a scalable, reliable, secure infrastructure. Systems management specialists certified in Tivoli® software from IBM, BMC, VeriSign and other products work with you to reduce total cost of ownership through systems optimization and business process transformation and to help increase return on your IT investment.

IBM **Infrastructure Resource Management Services** provide customers with a comprehensive solution for managing and supporting the technology infrastructure. From IBM expert consultants who can create and implement best-practices Infrastructure Resource Management strategies, to a full range of technology lifecycle out-tasking solutions that can augment customer staff and complement operations, the IBM approach provides the services necessary for success in the face of a rapidly changing e-business world. IBM professionals provide leadership and expertise for managing and supporting increasingly complex infrastructures through a combination of world-class skills, processes and tools. The service elements include consulting, procurement, configuration (DPP), rollouts/deployments, move/

add/change, tracking and reporting, wall-to-wall inventory, disposition and remote infrastructure support. Through our global product partners, Peregrine Systems Inc. and Remedy Corp., IBM provides expert help desk and asset management implementation services tied into all ISM services.

IBM **Performance Management, Testing and Scalability Services** help improve our customers' ability to deliver performance that their business and users demand. IBM provides comprehensive analysis and planning, with the skills and experience to help identify performance inhibitors and develop the best solution to meet current and future business needs. IBM services help customers support growth and a positive return on investment while delivering high-performance solutions for business productivity and planning for scalability and interoperability to support changing business demands.

### Summary

Autonomic computing defines the IT infrastructure ideal state. In addition to the IBM Global Services solutions and offerings described above, other new services (also already in place) can help an organization move rapidly toward the autonomic computing vision. **e-business on demand Services** provide a new way for customers to acquire infrastructure, applications and business processes over the Internet in an on-demand, pay-as-you-go model. **IT Optimization Services** deliver a holistic, multi-disciplinary, analytical approach for defining, prioritizing and implementing infrastructure-improvement initiatives that can help significantly increase infrastructure business value. **Resiliency Services** address the need for designing and implementing infrastructures that are flexible, highly resistant to the effects of a failure and capable of rapid recovery if failure does occur.

With these services and offerings as a foundation, IBM Global Services plans to continue to develop new, integrated services, methodologies and intellectual capital that, when linked with advanced technologies, can help our customers derive even more business value from their infrastructure investments.

For more information about IBM Global Services and autonomic computing, contact your IBM sales representative or visit **ibm.com**



IBM Corporation
Software Group
Route 100
Somers, NY 10589
U.S.A.

10-02


The IBM home page on the Internet can be found at **ibm.com**

G325-6919-00

# The dawning of the autonomic computing era

by A. G. Ganek
T. A. Corbi

This issue of the *IBM Systems Journal* explores a broad set of ideas and approaches to autonomic computing—some first steps in what we see as a journey to create more self-managing computing systems. Autonomic computing represents a collection and integration of technologies that enable the creation of an information technology computing infrastructure for IBM's agenda for the next era of computing—e-business on demand. This paper presents an overview of IBM's autonomic computing initiative. It examines the genesis of autonomic computing, the industry and marketplace drivers, the fundamental characteristics of autonomic systems, a framework for how systems will evolve to become more self-managing, and the key role for open industry standards needed to support autonomic behavior in heterogeneous system environments. Technologies explored in each of the papers presented in this issue are introduced for the reader.

On March 8, 2001, Paul Horn, IBM Senior Vice President and Director of Research, presented the theme and importance of autonomic computing to the National Academy of Engineering at Harvard University. His message was:

> The information technology industry loves to prove the impossible possible. We obliterate barriers and set records with astonishing regularity. But now we face a problem springing from the very core of our success—and too few of us are focused on solving it. More than any other I/T problem, this one—if it remains unsolved—will actually prevent us from moving to the next era of computing. The obstacle is complexity . . . Dealing with it is the single most important challenge facing the I/T industry.[1]

One month later, Irving Wladawsky-Berger, Vice President of Strategy and Technology for the IBM Server Group, introduced the Server Group's autonomic computing project (then named eLiza*[2]), with the goal of providing self-managing systems to address those concerns. Thus began IBM's commitment to deliver "autonomic computing"—a new company-wide and, it is to be hoped, industry-wide, initiative targeted at coping with the rapidly growing complexity of operating, managing, and integrating computing systems.

We do not see a change in Moore's law[3] that would slow development as the main obstacle to further progress in the information technology (IT) industry. Rather, it is the IT industry's exploitation of the technologies in accordance with Moore's law that has led to the verge of a complexity crisis. Software developers have fully exploited a four- to six-orders-of-magnitude increase in computational power—producing ever more sophisticated software applications and environments. There has been exponential growth in the number and variety of systems and

components. The value of database technology and the Internet has fueled significant growth in storage subsystems to hold petabytes[4] of structured and unstructured information. Networks have interconnected the distributed, heterogeneous systems of the IT industry. Our information society creates unpredictable and highly variable workloads on those networked systems. And today, those increasingly valuable, complex systems require more and more skilled IT professionals to install, configure, operate, tune, and maintain them.

IBM is using the phrase "autonomic computing"[5] to represent the vision of how IBM, the rest of the IT industry, academia, and the national laboratories can address this new challenge. By choosing the word "autonomic," IBM is making an analogy with the autonomic nervous system. The autonomic nervous system frees our conscious brain from the burden of having to deal with vital but lower-level functions. Autonomic computing will free system administrators from many of today's routine management and operational tasks. Corporations will be able to devote more of their IT skills toward fulfilling the needs of their core businesses, instead of having to spend an increasing amount of time dealing with the complexity of computing systems.

## Need for autonomic computing

As Frederick P. Brooks, Jr., one of the architects of the IBM System/360*, observed, "Complexity is the business we are in, and complexity is what limits us."[6] The computer industry has spent decades creating systems of marvelous and ever-increasing complexity. But today, complexity itself is the problem.

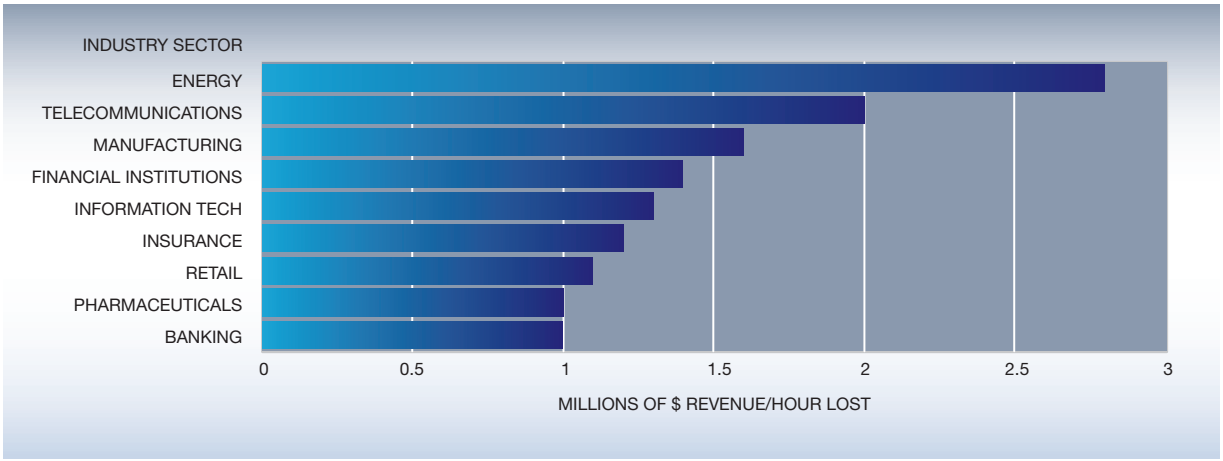
The spiraling cost of managing the increasing complexity of computing systems is becoming a significant inhibitor that threatens to undermine the future growth and societal benefits of information technology. Simply stated, managing complex systems has grown too costly and prone to error. Administering a myriad of system management details is too labor-intensive. People under such pressure make mistakes, increasing the potential of system outages with a concurrent impact on business. And, testing and tuning complex systems is becoming more difficult. Consider:

- It is now estimated that one-third to one-half of a company's total IT budget is spent preventing or recovering from crashes.[7]
- Nick Tabellion, CTO of Fujitsu Softek, said: "The commonly used number is: For every dollar to purchase storage, you spend $9 to have someone manage it."[8]
- Aberdeen Group studies show that administrative cost can account for 60 to 75 percent of the overall cost of database ownership (this includes administrative tools, installation, upgrade and deployment, training, administrator salaries, and service and support from database suppliers).[9]
- When you examine data on the root cause of computer system outages, you find that about 40 percent are caused by operator error,[10] and the reason is not because operators are not well-trained or do not have the right capabilities. Rather, it is because the complexities of today's computer systems are too difficult to understand, and IT operators and managers are under pressure to make decisions about problems in seconds.[11]
- A Yankee Group report[12] estimated that downtime caused by security incidents cost as much as $4,500,000 per hour for brokerages and $2,600,000 for banking firms.
- David J. Clancy, chief of the Computational Sciences Division at the NASA Ames Research Center, underscored the problem of the increasing systems complexity issues: "Forty percent of the group's software work is devoted to test," he said, and added, "As the range of behavior of a system grows, the test problem grows exponentially."[13]
- A recent Meta Group study looked at the impact of downtime by industry sector as shown in Figure 1.

Although estimated, cost data such as shown in Figure 1 are indicative of the economic impact of system failures and downtime. According to a recent IT resource survey by the Merit Project of Computer Associates International, 1867 respondents grouped the most common causes of outages into four areas of data center operations: systems, networks, database, and applications.[14] Most frequently cited outages included:

- For systems: operational error, user error, third-party software error, internally developed software problem, inadequate change control, lack of automated processes
- For networks: performance overload, peak load problems, insufficient bandwidth
- For database: out of disk space, log file full, performance overload
- For applications: application error, inadequate change control, operational error, nonautomated application exceptions

Figure 1 Downtime: Average hourly impact




Data from *IT Performance Engineering and Measurement Strategies: Quantifying Performance Loss*, Meta Group, Stamford, CT (October 2000).

Well-engineered autonomic functions targeted at improving and automating systems operations, installation, dependency management, and performance management can address many causes of these "most frequent" outages and reduce outages and downtime.

A confluence of marketplace forces are driving the industry toward autonomic computing. Complex heterogeneous infrastructures composed of dozens of applications, hundreds of system components, and thousands of tuning parameters are a reality. New business models depend on the IT infrastructure being available 24 hours a day, 7 days a week. In the face of an economic downturn, there is an increasing management focus on "return on investment" and operational cost controls—while staffing costs exceed the costs of technology. To compound matters further, there continues to be a scarcity of highly skilled IT professionals to install, configure, optimize, and maintain these complex, heterogeneous systems.

To respond, system design objectives must shift from the "pure" price/performance requirements to issues of robustness and manageability in the total-cost-of-ownership equation. As a profession, we must strive to simplify and automate the management of systems. Today's systems must evolve to become much more self-managing, that is: self-configuring, self-healing, self-optimizing, and self-protecting.

Irving Wladawsky-Berger outlined the solution at the Kennedy Consulting Summit in November 2001: "There is only one answer: The technology needs to manage itself. Now, I don't mean any far out AI project; what I mean is that we need to develop the right software, the right architecture, the right mechanisms . . . So that instead of the technology behaving in its usual pedantic way and requiring a human being to do everything for it, it starts behaving more like the 'intelligent' computer we all expect it to be, and starts taking care of its own needs. If it doesn't feel well, it does something. If someone is attacking it, the system recognizes it and deals with the attack. If it needs more computing power, it just goes and gets it, and it doesn't keep looking for human beings to step in."[15]

## What is autonomic computing?

Automating the management of computing resources is not a new problem for computer scientists. For decades system components and software have been evolving to deal with the increased complexity of system control, resource sharing, and operational management. Autonomic computing is just the next logical evolution of these past trends to address the increasingly complex and distributed computing environments of today. So why then is this something new? Why a call to arms to the industry for heightened focus and new approaches? The answer lies in the radical changes in the information technology environment in just the few short years since the mid-1990s, with the use of the Internet and e-business extending environments to a dramatically

larger scale, broader reach, and a more mission-critical fundamental requirement for business. In that time the norm for a large on-line system has escalated from applications such as networks consisting of tens of thousands of fixed-function automated teller machines connected over private networks to rich suites of financial services applications that can be accessed via a wide range of devices (personal computer, notebook, handheld device, smart phone, smart card, etc.) by tens of millions of people worldwide over the Internet.

IBM's autonomic computing initiative has been outlined broadly. Paul Horn[1] described this "grand challenge" and called for industry-wide collaboration toward developing autonomic computing systems that have characteristics as follows:

- To be autonomic, a system needs to "know itself"—and consist of components that also possess a system identity.
- An autonomic system must configure and reconfigure itself under varying and unpredictable conditions.
- An autonomic system never settles for the status quo—it always looks for ways to optimize its workings.
- An autonomic system must perform something akin to healing—it must be able to recover from routine and extraordinary events that might cause some parts to malfunction.
- A virtual world is no less dangerous than the physical one, so an autonomic computing system must be an expert in self-protection.
- An autonomic computing system knows its environment and the context surrounding its activity, and acts accordingly.
- An autonomic system cannot exist in a hermetic environment (and must adhere to open standards).
- Perhaps most critical for the user, an autonomic computing system will anticipate the optimized resources needed to meet a user's information needs while keeping its complexity hidden.

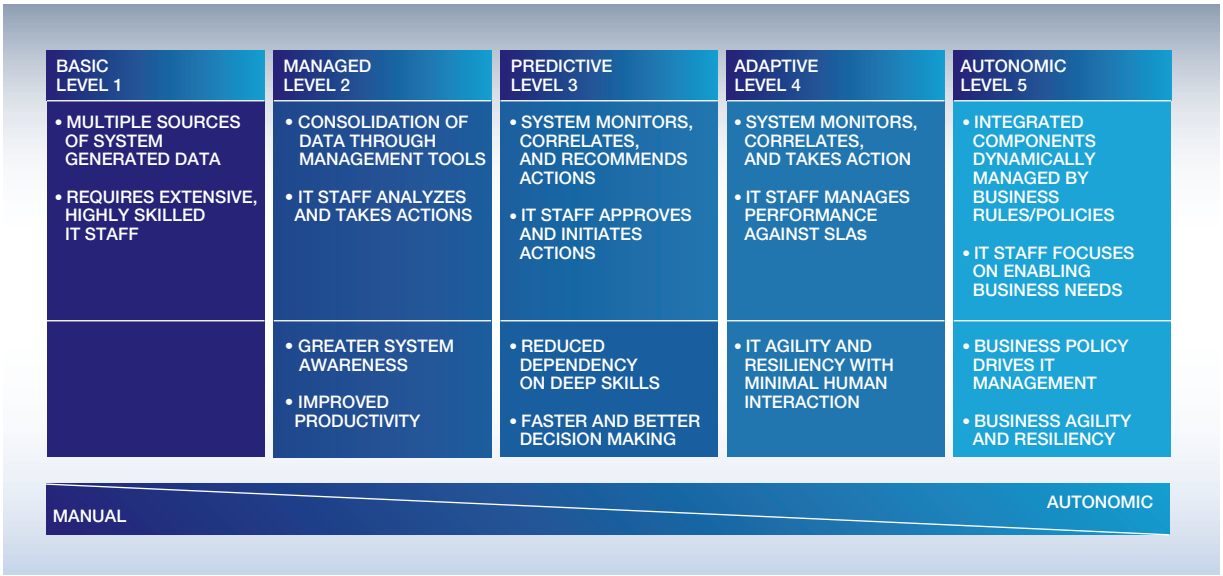
**Fundamentals of autonomic computing.** In order to incorporate these characteristics in "self-managing" systems, future autonomic computing systems will have four fundamental features. Various aspects of these four fundamental "self" properties are explored in this issue of the *IBM Systems Journal*.

***Self-configuring.*** Systems adapt automatically to dynamically changing environments. When hardware and software systems have the ability to define themselves "on-the fly," they are self-configuring. This aspect of self-managing means that new features, software, and servers can be dynamically added to the enterprise infrastructure with no disruption of services. Systems must be designed to provide this aspect at a feature level with capabilities such as plug and play devices, configuration setup wizards, and wireless server management. These features will allow functions to be added dynamically to the enterprise infrastructure with minimum human intervention. Self-configuring not only includes the ability for each individual system to configure itself on the fly, but also for systems within the enterprise to configure themselves into the e-business infrastructure of the enterprise. The goal of autonomic computing is to provide self-configuration capabilities for the entire IT infrastructure, not just individual servers, software, and storage devices.

***Self-healing.*** Systems discover, diagnose, and react to disruptions. For a system to be self-healing, it must be able to recover from a failed component by first detecting and isolating the failed component, taking it off line, fixing or isolating the failed component, and reintroducing the fixed or replacement component into service without any apparent application disruption. Systems will need to predict problems and take actions to prevent the failure from having an impact on applications. The self-healing objective must be to minimize all outages in order to keep enterprise applications up and available at all times. Developers of system components need to focus on maximizing the reliability and availability design of each hardware and software product toward continuous availability.

***Self-optimizing.*** Systems monitor and tune resources automatically. Self-optimization requires hardware and software systems to efficiently maximize resource utilization to meet end-user needs without human intervention. IBM systems already include industry-leading technologies such as logical partitioning, dynamic workload management, and dynamic server clustering. These kinds of capabilities should be extended across multiple heterogeneous systems to provide a single collection of computing resources that could be managed by a "logical" workload manager across the enterprise. Resource allocation and workload management must allow dynamic redistribution of workloads to systems that have the necessary resources to meet workload requirements. Similarly, storage, databases, networks, and other resources must be continually tuned to enable efficient operations even in unpredictable environments. Fea-

Figure 2 Evolving to autonomic operations




From *IBM Global Services and Autonomic Computing*, IBM White Paper, October 2002; see http://www-3.ibm.com/autonomic/pdfs/wp-igs-autonomic.pdf.

tures must be introduced to allow the enterprise to optimize resource usage across the collection of systems within their infrastructure, while also maintaining their flexibility to meet the ever-changing needs of the enterprise.

***Self-protecting.*** Systems anticipate, detect, identify, and protect themselves from attacks from anywhere. Self-protecting systems must have the ability to define and manage user access to all computing resources within the enterprise, to protect against unauthorized resource access, to detect intrusions and report and prevent these activities as they occur, and to provide backup and recovery capabilities that are as secure as the original resource management systems. Systems will need to build on top of a number of core security technologies already available today, including LDAP (Lightweight Directory Access Protocol), Kerberos, hardware encryption, and SSL (Secure Socket Layer). Capabilities must be provided to more easily understand and handle user identities in various contexts, removing the burden from administrators.

## An evolution, not a revolution

To implement autonomic computing, the industry must take an evolutionary approach and deliver improvements to current systems that will provide significant self-managing value to customers without requiring them to completely replace their current IT environments. New open standards must be developed that will define the new mechanisms for interoperating heterogeneous systems. Figure 2 is a representation of those levels, starting from the basic level, through managed, predictive, and adaptive levels, and finally to the autonomic level.

As seen in the figure, the basic level represents the starting point where some IT systems are today. Each system element is managed independently by IT professionals who set it up, monitor it, and eventually replace it. At the managed level, systems management technologies can be used to collect information from disparate systems onto fewer consoles, reducing the time it takes for the administrator to collect and synthesize information as the systems become more complex to operate. In the predictive level, as new technologies are introduced that provide correlation among several elements of the system, the system itself can begin to recognize patterns, predict the optimal configuration, and provide advice on what course of action the administrator should take. As these technologies improve and as people become more comfortable with the advice and predictive power of these systems, we can pro-

Table 1 Aligning with business goals

| Basic | Managed | Predictive | Adaptive | Autonomic |
|---|---|---|---|---|
| **Process:** Informal, reactive, manual | **Process:** Documented, improved over time, leverage of industry best practices; manual process to review IT performance | **Process:** Proactive, shorter approval cycle | **Process:** Automation of many resource mgmt best practices and transaction mgmt best practices, driven by service level agreements | **Process:** All IT service mgmt and IT resource mgmt best practices are automated |
| **Tools:** Local, platform- and product-specific | **Tools:** Consolidated resource mgmt consoles with problem mgmt system, automated software install, intrusion detection, load balancing | **Tools:** Role-based consoles with analysis and recommendations; product configuration advisors; real-time view of current & future IT performance; automation of some repetitive tasks; common knowledge base of inventory and dependency management | **Tools:** Policy management tools that drive dynamic change based on resource specific policies | **Tools:** Costing/financial analysis tools, business and IT modeling tools, trade-off analysis; automation of some e-business mgmt roles |
| **Skills:** Platform-specific, geographically dispersed with technology | **Skills:** Multiple skill levels with centralized triage to prioritize and assign problems to skilled IT professionals | **Skills:** Cross-platform system knowledge, IT workload management skills, some bus process knowledge | **Skills:** Service objectives and delivery per resource, and analysis of impact on business objectives | **Skills:** E-business cost & benefit analysis, performance modeling, advanced use of financial tools for IT context |
| **Benchmarks:** Time to fix problems and finish tasks | **Benchmarks:** System availability, time to close trouble tickets and work requests | **Benchmarks:** Business system availability, service level agreement attainment, customer satisfaction | **Benchmarks:** Business system response time, service level agreement attainment, customer satisfaction, IT contribution to business success | **Benchmarks:** Business success, competitiveness of service level agreement metrics, business responsiveness |

Source: *Autonomic Computing Concepts*, IBM White Paper, October 2002; see: http://www-3.ibm.com/autonomic/pdfs/AC_Concepts.pdf.

gress to the adaptive level where the systems themselves can automatically take the correct actions based on the information that is available to them and the knowledge of what is happening in the systems. Service Level Agreements (SLAs)[16] guide operation of the system. Finally, at the fully autonomic level, the system operation is governed by business policies and objectives. Users interact with the system to monitor the business processes or alter the objectives.

As companies progress through the five levels of autonomic computing, the processes, tools, and benchmarks become increasingly sophisticated, and the skills requirement becomes more closely aligned with the business. Table 1 illustrates this correlation.

The basic level represents the starting point for most IT organizations. If they are formally measured at all, they are typically measured on the time required to finish major tasks and fix major problems. The IT organization is viewed as a cost center, in which the variable costs associated with labor are preferred over an investment in centrally coordinated systems management tools and processes.

At the managed level, IT organizations are measured on the availability of their managed resources, their

time to close trouble tickets in their problem management system, and their time to complete formally tracked work requests. To improve on these measurements, IT organizations document their processes and continually improve them through manual feedback loops and adoption of best practices. IT organizations gain efficiency through consolidation of management tools to a set of strategic platforms and through a hierarchical problem management triage organization.

In the predictive level, IT organizations are measured on the availability and performance of their business systems and their return on investment. To improve on these measurements, IT organizations measure, manage, and analyze transaction performance. The implications of the critical nature of the role of the IT organization in the success of the business are understood. Predictive tools are used to project future IT performance, and many tools make recommendations to improve future performance.

In the adaptive level, IT resources are automatically provisioned and tuned to optimize transaction performance. Business policies, business priorities, and service-level agreements guide the autonomic infrastructure behavior. IT organizations are measured on end-to-end business system response times (transaction performance), the degree of efficiency with which the IT infrastructure is utilized, and their ability to adapt to shifting workloads.

In the autonomic level, IT organizations are measured on their ability to make the business successful. To improve business measurements, IT tools understand the financial metrics associated with e-business activities and supporting IT activities. Advanced modeling techniques are used to optimize e-business performance and quickly deploy newly optimized e-business solutions.

Today's software and hardware system components will evolve toward more autonomic behavior. For example:

- Data management. New database software tools can use statistics from the databases, analyze them, and learn from the historical system performance information. The tools can help an enhanced database system automatically detect potential bottlenecks as they are about to occur and attempt to compensate for them by adjusting tuning parameters. Query optimizers can learn the optimal index and route to certain data and automatically seek out that path based on the historical access patterns and associated response times.

- Web servers and software. Web servers can provide real-time diagnostic "dashboard" information, enabling customers to more quickly become aware of resource problems, instead of relying on after-the-fact reports to identify problems. Once improved instrumentation is available, autonomic functions can be introduced that enable the Web server infrastructure to automatically monitor, analyze, and fix performance problems. As an example, suppose an application server is freezing-up intermittently, and no customer transactions are being processed for several seconds, thus losing thousands of dollars in business, as well as customer confidence and loyalty. Using real-time monitoring, predictive analysis, and auto-tuning, the freeze-up is anticipated before it happens. The autonomic function compares real-time data with historical problem data (i.e., suggesting that the cache sizes were set too low). The settings are reset automatically without service disruption, and a report is sent to the administrator that shows what action was taken.

- Systems management. Systems management software can contain improved problem determination and data collection features designed to help businesses better diagnose and prevent interruptions (or breaches of security). Such systems management software must enable customers to take an "end-to-end" view of their computing environment across multiple, independently installed hardware and software elements. A bank transaction, for example, might "touch" a discrete database, another transaction, and Web application servers as it is processed across a network. If a problem occurs with processing on one of the individual components, lack of an integrated problem determination infrastructure makes it more difficult to determine what prevented that bank transaction from completing successfully. A consolidated view created by the system management software would enable the system and IT staffs to identify and quickly react to problems as they happen by providing an end-to-end view of the application. The end-to-end view of the environment allows companies to understand problems and performance information in the context of their business goals.

- Servers. Computers can be built that need less human supervision. Computers can try to fix themselves in the event of a failure, protect themselves

from hacker attacks, and configure themselves when adding new features. Servers can use software algorithms that learn patterns in Internet traffic or application usage, and provision resources in a way that gives the shortest response time to the task with the highest business priority. Server support for heterogeneous and enterprise workload management, dynamic clustering, dynamic partitioning, improved setup wizards, improved user authentication, directory integration, and other tools to protect access to network resources are all steps toward more autonomic functioning.

IBM hardware and software systems have already made significant progress in introducing autonomic computing functionality.[2] But there is much more work ahead. The efforts to achieve cohesive system behavior must go beyond improvements in the individual components alone. These components must be federated, employing an integrating architecture that establishes the instrumentation, policy, and collaboration technologies so that groups of resources can work in concert, as for example, across systems in a grid. System management tools will play a central role in coordinating the actions of system components, providing a simplified mechanism for system administration and for translating business objectives into executable policies to govern the actions of the IT resources available.

## Industry standards are needed to support autonomic computing

Most IT infrastructures are composed of components supplied by different vendors. Open industry standards are the key to the construction of autonomic computing systems. Systems will need more standardization to introduce a uniform approach to instrumentation and data collection, dynamic configuration, and operation. Uniformity will allow the intersystem exchange of instrumentation and control information to create the basis for collaboration and autonomic behavior among heterogeneous systems.

For example, in storage systems, a standard that has been proposed for specifying data collection items is the Bluefin specification. Bluefin[17] defines a language and schema that allow users to reliably identify, classify, monitor, and control the physical and logical devices in storage area networking. The Storage Networking Industry Association (SNIA) has taken this standard to the Distributed Management Task Force (DMTF). SNIA is using Bluefin as the basis for its storage management initiative, the intent of which is to become the SNIA standard for management.

In the case of application instrumentation, the standard that has been proposed for obtaining the transaction rate, response time, failure rate, and topology data from applications is the Open Group Application Response Measurement (ARM)[18] application programming interfaces (APIs). The Application Response Measurement API defines the function calls that can be used to instrument an application or other software for transaction monitoring. It provides a way to monitor business transactions by embedding simple cells in the software that can be captured by an agent supporting the ARM API. The calls are used to capture data, allowing software to be monitored for availability, service levels, and capacity.

Other standards, such as the DMTF Common Information Model (CIM)[19] and Web Service Level Agreement (WSLA), provide languages and schemas for defining the available data. CIM is an object-oriented information model that provides a conceptual view of physical and logical system components. WSLA is a language to express SLA contracts, to support guaranteed performance, and to handle complex dynamic fluctuations in service demand. SLA-based system management would enable service providers to offer the same Web service at different performance levels, depending on contracts with their customers. WSLA is available through the IBM alphaWorks* Web Services Toolkit[20] that features a WSLA document approach based on Extensible Markup Language (XML) to define SLAs.

These standards are technologies that enable the building of "inter-communicating" autonomic system elements that are the foundation for cooperation in a federation of system components. Each individual autonomic "element" is responsible for managing itself, that is, for configuring itself internally, for healing internal failures when possible, for optimizing its own behavior, and for protecting itself from external probing and attack.

Autonomic elements are the building blocks for making autonomic systems. Autonomic elements continuously monitor system (or component) behavior through "sensors" and make adjustments through "effectors." By monitoring behavior through sensors, analyzing those data, then planning what action should be taken next (if any), and executing that ac-

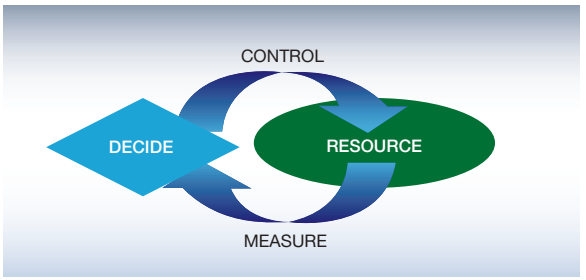
tion through effectors, a kind of "control loop"[21] is created (see Figure 3).

Interconnecting autonomic elements requires distributed computing mechanisms to access resources across the network. "Grid computing"[22] encompasses the idea of an emerging infrastructure that is focused on networking together heterogeneous, multiple regional and national computing systems. It has been called the next evolutionary step for the Internet. The term "grid" was chosen as an analogy with the electric power grid, which supplies pervasive access to power. Grids are persistent computing environments that enable software applications to integrate instruments, displays, and computational and information resources that are managed by diverse organizations in widespread locations.

In 2001, the Globus Project[23,24] launched a research and development program aimed at creating a toolkit based on the Open Grid Service Architecture (OGSA) that defines standard mechanisms for creating, naming, and discovering services and specifies various protocols to support accessing services. Essentially, OGSA is a framework for distributed computing, based on Web services protocols. Although OGSA is a proposed standard that will be developed and defined in the Global Grid Forum (GGF),[25] it is applicable whether the environment consists of a multiorganization grid or simply distributed resources within an enterprise. IBM, Microsoft Corporation, and others have already announced support for the OGSA framework. Work efforts on grid and OGSA are creating important architectural models and new open industry standards that are enablers for the IT industry to make progress toward more self-managing systems. Since grid deployments can expand the domain of computing across many systems, in our view, a successful grid system will require autonomic functionality.

Individual autonomic elements can interact through OGSA mechanisms. For example, today there is no accepted "sensor and effector" standard. But, the Globus Toolkit provides information services utilities to provide information about the status of grid resources. One of these utilities is the Monitoring and Discovery Service (MDS),[26] MDS 2.2 GRIS "Information Providers" that are essentially sensors or probes. The Globus Toolkit also provides a mechanism for authenticated access to MDS. Fault detection allows a client process to be monitored by a heartbeat monitor. Resource management APIs provide some job management capabilities.

Figure 3 Control loop




From *Autonomic Computing Concepts*, IBM White Paper, 2001

Thus we are seeing the emergence of some basic standard mechanisms needed for distributed "control loops" that in turn are needed for autonomic computing. When control loop standards are in place, the industry must address the more complex issues of specifying and automating policy management and service level agreements (SLAs).

A typical enterprise has a heterogeneous set of routers, firewalls, Web servers, databases, and workstations, all with different system management mechanisms. So again, industry standards will be needed in order to enable true policy management. We expect that policy specifications will be widely used in enterprises for defining quality of service management, storage backup, and system configuration, as well as security authorization and management.

A common approach to specifying and deploying policy would enable an enterprise to define and disseminate policies that reflect its overall IT service goals. A common, standard set of tools and techniques used throughout the enterprise could simplify analysis and reduce inconsistencies and conflicts in the policies deployed across the various components within the enterprise and also allow a policy exchange with external service providers.

Various standards bodies are working on specifying policies for network and systems management, security, and role-based access control (RBAC). The Internet Engineering Task Force (IETF)[27] and DMTF[28] have been concentrating on information models for management policies, protocols for transferring policies to network devices, and routing policies; the National Institute of Standards and Technology (NIST)[29] is working toward an RBAC standard; and the Oasis consortium (Organization for the Advancement of

Structured Information Standards)[30] is working on an XML-based specification of access control policies and authentication information.

It will take some time for the current divergent standards policy-based solutions to come to embrace a common approach. Meanwhile, research on policy-based management approaches continues.[31,32] Advances in policy management are needed to enable enterprises to eventually specify the behaviors of IT services in terms of the business process objectives of the enterprises.

## Exploratory research and development presented in this issue

Autonomic computing represents an exciting new research direction in computing. IBM believes that meeting the grand challenge of autonomic computing systems will involve researchers in a diverse array of fields, including systems management, distributed computing, networking, operations research, software development, storage, artificial intelligence, and control theory, as well as others.

The challenge of autonomic computing requires more than the re-engineering of today's systems. Autonomic computing also requires new ideas, new insights, and new approaches. This issue of the *IBM Systems Journal* provides just a glimpse into an array of research and development efforts underway for autonomic computing. Below we present the topics in the issue.

D. C. Verma, S. Sahu, S. Calo, A. Shaikh, I. Chang, and A. Acharya in their paper, "SRIRAM: A Scalable Resilient Autonomic Mesh,"[33] propose a method that facilitates instantiating mirroring and replication of services in a network of servers.

The ability to redistribute hardware resources dynamically is essential to both the self-configuring and self-optimizing goals of autonomic computing. J. Jann, L. M. Browning, and R. S. Burugula describe this new server capability in "Dynamic Reconfiguration: Basic Building Blocks for Autonomic Computing on IBM pSeries Servers."[34]

In the first of two invited papers, D. A. Norman and A. Ortony from Northwestern University, along with D. M. Russell of IBM, discuss in "Affect and Machine Design: Lessons for the Development of Autonomous Machines"[35] how studying the human characteristics of cognition and affect will help designers in developing complex autonomic systems that will interact with unpredictable situations.

K. Whisnant, Z. T. Kalbarczyk, and R. K. Iyer examine the difficulties of dynamically reconfiguring application software in their paper, "A System Model for Dynamically Reconfigurable Software."[36] They believe that both static structure and run-time behaviors must be captured in order to define a workable reconfiguration model.

One technology to support self-healing and self-configuring is the ability to dynamically insert new pieces of software and remove other pieces of code, without shutting down the running system. This technology is being explored in the K42 research operating system and is presented in the paper by J. Appavoo, K. Hui, C. A. N. Soules, R. W. Wisniewski, D. M. Da Silva, O. Krieger, M. A. Auslander, D. J. Edelsohn, B. Gamsa, G. R. Ganger, P. McKenney, M. Ostrowski, B. Rosenburg, M. Stumm, and J. Xenidis, entitled "Enabling Autonomic Behavior in Systems Software with Hot Swapping."[37]

L. W. Russell, S. P. Morgan, and E. G. Chron introduce the idea of a predictive autonomic system in their paper entitled "Clockwork: A New Movement in Autonomic Systems."[38] They explore the idea of a system that anticipates workload needs based on statistical modeling, tracking, and forecasting.

Component-based development, where multiple distributed software components are composed to deliver a particular business function, is an emerging programming model in the Web services world. D. M. Yellin in his paper, "Competitive Algorithms for the Dynamic Selection of Component Implementations,"[39] proposes a strategy and framework for optimizing component performance based on switching between different component implementations.

In an example of "self-optimizing," V. Markl, G. M. Lohman, and V. Raman discuss improving query performance by comparing estimates with actual results toward self-validating query planning in "LEO: An Autonomic Query Optimizer for DB2."[40]

As noted, system and network security are fundamental to autonomic computing systems. In "Security in an Autonomic Computing Environment,"[41] D. M. Chess, C. C. Palmer, and S. R. White outline a number of security and privacy issues in the design and development of autonomic systems.

G. Lanfranchi, P. Della Peruta, A. Perrone, and D. Calvanese describe what they see as a paradigm shift in system management needed for autonomic computing. In their paper, "Toward a New Landscape of Systems Management in an Autonomic Computing Environment,"[42] they introduce a knowledge-based resource model technology that extends across design, delivery, and run time.

In the second invited paper, "Comparing Autonomic and Proactive Computing,"[43] R. Want, T. Pering, and D. Tennenhouse of Intel Research present a high-level discussion of the similarities between proactive computing and autonomic computing with an emphasis on their research in proactive computing—an environment in which computers anticipate what users need and act accordingly.

Today, optimizing performance in multisystem e-commerce environments requires considerable skill and experience. In "Managing Web Server Performance with AutoTune Agents,"[44] Y. Diao, J. L. Hellerstein, S. Parekh, and J. P. Bigus describe intelligent agents that use control theory techniques to autonomically adjust an Apache** Web server to dynamic workloads.

The backbone of a grid or typical autonomic computing system will be an intelligent, heterogeneous network infrastructure. Management issues related to topology, service placement, cost and service metrics, as well as dynamic administration structure are explored by R. Haas, P. Droz, and B. Stiller in "Autonomic Service Deployment in Networks."[45]

Although much of the discussion on autonomic computing often focuses on servers, networks, databases, and storage management, we realize that personal computer users would also benefit greatly by the introduction of autonomic features. D. F. Bantz, C. Bisdikian, D. Challener, J. P. Karidis, S. Mastrianni, A. Mohindra, D. G. Shea, and M. Vanover explore these possibilities in their paper, "Autonomic Personal Computing."[46]

People will still need to interact with autonomic computing systems. D. M. Russell, P. P. Maglio, R. Dordick, and C. Neti in their paper entitled "Dealing with Ghosts: Managing the User Experience of Autonomic Computing"[47] argue that the lessons we have learned in human-computer interaction research must be applied to effectively expose and communicate the run-time behavior of these complex systems and to better define and structure the user system operation scenarios.

In the Technical Forum section, the complex challenges of life-cycle management and providing capacity on demand are examined in a project as described by A. Abbondanzio, Y. Aridor, O. Biran, L. L. Fong, G. S. Goldszmidt, R. E. Harper, S. M. Krishnakumar, G. Pruett, and B.-A. Yassur in "Management of Application Complexes in Multitier Clustered Systems."[48]

## Conclusion

In his keynote speech at the Almaden Institute 2002,[49] John Hennessy, President of Stanford University, presented his view of the autonomic challenge. While acknowledging the significant accomplishments in hardware architecture over the past 20 years, he urged industry and academia to look forward and to shift focus to a set of issues related to how services will be delivered over networks in the Internet/Web-centric "post-desktop" era: "As the business use of this environment grows and as people become more and more used to it, the flakiness that we've all accepted in the first generation of the Internet and the Web—will become unacceptable." Hennessy emphasized an increased research focus on availability, maintainability, scalability, cost, and performance—all fundamental aspects of autonomic computing.

Autonomic computing is a journey. Progress will be made in a series of evolutionary steps. This issue of the *IBM Systems Journal* presents some of the technology signposts that can serve to guide the ongoing research in this new direction.

## Acknowledgments

First, we express our thanks to the many authors who submitted their work for publication in this issue. Special thanks go to Lorraine Herger, Kazuo Iwano, Pratap Pattnaik, Connie Marconi, and Felicia C. Medley, who organized the call for papers, managed the process to select the topics and the papers, and worked with the authors and *IBM Systems Journal* staff to produce this issue. We also thank the numerous referees whose comments helped all the authors in improving their papers.

We particularly express our appreciation to Paul Horn, IBM Senior Vice President and Director of Research, for launching the autonomic computing

grand challenge in his 2001 presentation on autonomic computing and to Irving Wladawsky-Berger who led the launch of the autonomic computing project in the IBM Server Group.

We acknowledge the efforts of Robert Morris, Anant Jhingran, Tushar Chandra, and K. M. Moiduddin, who organized the Almaden Autonomic Computing Institute held at San Jose, California in April 2002. We also acknowledge Sam S. Adams, Lisa F. Spainhower, William H. Tetzlaff, William H. Chung, Steve R. White, and Kazuo Iwano who organized the Autonomic Computing Conference sponsored by the IBM Academy of Technology and held in Yorktown Heights, New York in May 2002. Our thanks go to Christopher W. Luongo for his article, "Server to IT: Thanks, But I Fixed It Myself."

The authors would like to thank Ric Telford, John Sweitzer, and Jim Crosskey of the Autonomic Computing department for their contributions to this manuscript. The authors also wish to acknowledge the *IBM Systems Journal* staff for their many helpful suggestions during the creation of this issue.

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of Apache Digital Corporation.

## Cited references and notes

1. P. Horn, *Autonomic Computing: IBM's Perspective on the State of Information Technology*, IBM Corporation (October 15, 2001); available at http://www.research.ibm.com/autonomic/manifesto/autonomic_computing.pdf.
2. IBM Server Group, *eLiza: Building an Intelligent Infrastructure for E-business—Technology for a Self-Managing Server Environment*, G520-9592-00, IBM Corporation (2001); also at http://www-1.ibm.com/servers/eserver/introducing/eliza/eliza_final.pdf.
3. For more than 30 years, Moore's Law has forecasted progress in the computing industry like an immutable force of nature. In 1965, Gordon E. Moore, then the director of research and development at Fairchild Semiconductor Corporation, made the casual observation that processing power will double every 18 to 24 months, suggesting healthy growth ahead over the next decade for the then-nascent silicon chip industry. Five years later, Moore cofounded Intel Corporation, and "Moore's law" was well on its way to becoming a self-fulfilling prophecy among researchers and developers in the industry.
4. A petabyte (PB) is 1000 terabytes, and a terabyte is 1000 gigabytes. CERN, the European Center for Nuclear Research located just outside of Geneva, Switzerland, has started building a 100-PB archive to house data from the particle accelerator of the research center. See "From Kilobytes to Petabytes in 50 Years," *Science and Technology Review*, UCRL-52000-02-4, Lawrence Livermore National Laboratory, Livermore, CA (April 15, 2002), http://www.llnl.gov/str/March02/March50th.html.
5. The term "autonomic computing" comes from an analogy to the autonomic central nervous system in the human body, which adjusts to many situations without any external help. "Autonomic" is defined as: (a) Of, relating to, or controlled by the autonomic nervous system; (b) Acting, or occurring involuntarily; automatic; an autonomic reflex.
6. F. P. Brooks, Jr., *The Mythical Man-Month: Essays on Software Engineering*, Twentieth Anniversary Edition, Addison-Wesley Publishing Co., Reading, MA (1995), p. 226. See also, F. P. Brooks, Jr., "No Silver Bullet: Essence and Accidents of Software Engineering," *Computer* **20**, No. 4, 10–19 (1987).
7. D. A. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kiciman, M. Merzbacher, D. Oppenheimer, N. Sastry, W. Tetzlaff, J. Traupman, N. Treuhaft, *Recovery-Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies*, U.C. Berkeley Computer Science Technical Report, UCB//CSD-02-1175, University of California, Berkeley (March 15, 2002).
8. K. Evans-Correia, "Simplifying Storage Management Starts with More Efficient System Utilization," Interview with N. Tabellion, *searchStorage* (August 29, 2001), see http://searchstorage.techtarget.com/qna/0,289202,sid5_gci764063,00.html.
9. *IBM Data Management Tools: New Opportunities for Cost-Effective Administration*, Profile Report, Aberdeen Group, Inc., Boston (April 2002), p. 3.
10. D. Patterson, "Availability and Maintainability >> Performance: New Focus for a New Century," *USENIX Conference on File and Storage Technologies* (FAST '02), Keynote Address, Monterey, CA (January 29, 2002).
11. A. Brown and D. A. Patterson, "To Err Is Human," *Proceedings of the First Workshop on Evaluating and Architecting System dependabilitY* (EASY '01), Goeteborg, Sweden (July 2001).
12. "How Much Is an Hour of Downtime Worth to You?" from *Must-Know Business Continuity Strategies*, Yankee Group, Boston (July 31, 2002).
13. D. J. Clancy, "NASA Challenges in Autonomic Computing," *Almaden Institute 2002*, IBM Almaden Research Center, San Jose, CA (April 10, 2002).
14. Merit Project, Computer Associates International, http://www.meritproject.com/it_survey_results.htm.
15. I. Wladawsky-Berger, "Advancing E-business into the Future: The Grid," *Kennedy Consulting Summit 2001*, New York (November 29, 2001).
16. In this context, a Service Level Agreement (SLA) is a compact between a customer or consumer and a provider of an IT service that specifies the levels of availability, serviceability, performance (and tracking/reporting), problem management, security, operation, or other attributes of the service, often established via negotiation. Typically, an SLA identifies and defines the customer's needs, provides a framework for discussion and understanding, attempts to simplify complex requirements, outlines methods for resolving disputes, and helps eliminate unrealistic expectations.
17. "'Bluefin' A Common Interface for SAN Management," White Paper, Storage Networking Industry Association (August 13, 2002), http://www.snia.org/tech_activities/SMI/bluefin/Bluefin_White_Paper_v081302.pdf from http://www.snia.org/tech_activities/SMI/bluefin/.
18. *Application Response Measurement Issue 3.0 Java Binding*, Open Group Technical Standard CO14, The Open Group (October 2001), at http://www.opengroup.org/products/publications/catalog/c014.htm.

19. *Common Information Model (CIM) Specification Version 2.2*, DSP0004, Distributed Management Task Force (June 14, 1999), at http://www.dmtf.org/standards/standard_cim.php.
20. Web Services Toolkit, alphaWorks, IBM Corporation (July 26, 2000), http://www.alphaworks.ibm.com/tech/webservices toolkit.
21. The control loop is the essence of automation. By measuring or sensing some activity in a process to be controlled, a controller component decides what needs to be done next and executes the required operations through a set of actuators. The controller then remeasures the process to determine whether the actions of the actuator had the desired effect. The whole routine is then repeated in a continuous loop of measure, decide, actuate, and repeat.
22. *The Grid: Blueprint for a New Computing Infrastructure*, I. Foster and C. Kesselman, Editors, Morgan Kaufmann Publishers, Inc., San Francisco, CA (1999).
23. See http://www.globus.org, the home of the Globus Project, the Globus Toolkit (GT3), and work related to the Open Grid Services Architecture (OGSA).
24. I. Foster, C. Kesselman, and S. Tuecke, "The Anatomy of the Grid: Enabling Scalable Virtual Organizations," *International Journal of High Performance Computing* **15**, No. 3, 200–222 (2001); see also, http://www.globus.org/research/papers/anatomy.pdf.
25. Global Grid Forum, http://www.gridforum.org/.
26. *MDS 2.2 User's Guide*, The Globus Project, available at www.globus.org/mds/mdsusersguide.pdf.
27. Internet Engineering Task Force, http://www.dmtf.org.
28. Distributed Management Task Force, http://www.dmtf.org.
29. National Institute of Standards and Technology, http://www.nist.org.
30. Organization for the Advancement of Structured Information Standards, http://www.oasis-open.org.
31. L. Lymberopoulos, E. Lupu, and M. Sloman, "An Adaptive Policy Based Management Framework for Differentiated Services Networks," *Proceedings of the 3rd IEEE Workshop on Policies for Distributed Systems and Networks (Policy 2002)*, Monterey, CA (June 2002), pp. 147–158.
32. V. Sander, W. A. Adamson, I. Foster, and A. Roy, "End-to-End Provision of Policy Information for Network QoS," *Proceedings of the Tenth IEEE Symposium on High Performance Distributed Computing* (HPDC-10), IEEE Press (August 2001).
33. D. C. Verma, S. Sahu, S. Calo, A. Shaikh, I. Chang, and A. Acharya, "SRIRAM: A Scalable Resilient Autonomic Mesh," *IBM Systems Journal* **42**, No. 1, 19–28 (2003, this issue).
34. J. Jann, L. A. Browning, and R. S. Burugula, "Dynamic Reconfiguration: Basic Building Blocks for Autonomic Computing on IBM pSeries Servers," *IBM Systems Journal* **42**, No. 1, 29–37 (2003, this issue).
35. D. A. Norman, A. Ortony, and D. M. Russell, "Affect and Machine Design: Lessons for the Development of Autonomous Machines," *IBM Systems Journal* **42**, No. 1, 38–44 (2003, this issue).
36. K. Whisnant, Z. T. Kalbarczyk, and R. K. Iyer, "A System Model for Dynamically Reconfigurable Software," *IBM Systems Journal* **42**, No. 1, 45–59 (2003, this issue).
37. J. Appavoo, K. Hui, C. A. N. Soules, R. W. Wisniewski, D. M. Da Silva, O. Krieger, M. A. Auslander, D. J. Edelsohn, B. Gamsa, G. R. Ganger, P. McKenney, M. Ostrowski, B. Rosenburg, M. Stumm, and J. Xenidis, "Enabling Autonomic Behavior in Systems Software with Hot Swapping," *IBM Systems Journal* **42**, No. 1, 60–76 (2003, this issue).
38. L. W. Russell, S. P. Morgan, and E. G. Chron, "Clockwork: A New Movement in Autonomic Systems," *IBM Systems Journal* **42**, No. 1, 77–84 (2003, this issue).
39. D. M. Yellin, "Competitive Algorithms for the Dynamic Selection of Component Implementations," *IBM Systems Journal* **42**, No. 1, 85–97 (2003, this issue).
40. V. Markl, G. M. Lohman, and V. Raman, "LEO: An Autonomic Query Optimizer for DB2," *IBM Systems Journal* **42**, No. 1, 98–106 (2003, this issue).
41. D. M. Chess, C. C. Palmer, and S. R. White, "Security in an Autonomic Computing Environment," *IBM Systems Journal* **42**, No. 1, 107–118 (2003, this issue).
42. G. Lanfranchi, P. Della Peruta, A. Perrone, and D. Calvanese, "Toward a New Landscape of Systems Management in an Autonomic Computing Environment," *IBM Systems Journal* **42**, No. 1, 119–128 (2003, this issue).
43. R. Want, T. Pering, and D. Tennenhouse, "Comparing Autonomic and Proactive Computing," *IBM Systems Journal* **42**, No. 1, 129–135 (2003, this issue).
44. Y. Diao, J. L. Hellerstein, S. Parekh, and J. P. Bigus, "Managing Web Server Performance with AutoTune Agents," *IBM Systems Journal* **42**, No. 1, 136–149 (2003, this issue).
45. R. Haas, P. Droz, and B. Stiller, "Autonomic Service Deployment in Networks," *IBM Systems Journal* **42**, No. 1, 150–164 (2003, this issue).
46. D. F. Bantz, C. Bisdikian, C. Challener, J. P. Karidis, S. Mastrianni, A. Mohindra, D. G. Shea, and M. Vanover, "Autonomic Personal Computing," *IBM Systems Journal* **42**, No. 1, 165–176 (2003, this issue).
47. D. M. Russell, P. P. Maglio, R. Dordick, and C. Neti, "Dealing with Ghosts: Managing the User Experience of Autonomic Computing," *IBM Systems Journal* **42**, No. 1, 177–188 (2003, this issue).
48. A. Abbondanzio, Y. Aridor, O. Biran, L. L. Fong, G. S. Goldszmidt, R. E. Harper, S. M. Krishnakumar, G. Pruett, and B.-A. Yassur, "Management of Application Complexes in Multitier Clustered Systems," Technical Forum, *IBM Systems Journal* **42**, No. 1, 189–195 (2003, this issue).
49. J. Hennessy, "Back to the Future: Time to Return to Some Long-Standing Problems in Computer Science," *Almaden Institute 2002*, IBM Almaden Research Center, San Jose, CA (April 10, 2002).

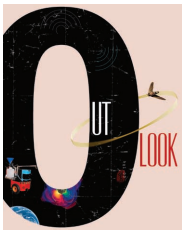
*Accepted for publication November 1, 2002.*

**Alan G. Ganek** *IBM Software Group, Hawthorne II, 17 Sky Line Drive, Hawthorne, New York 10532 (electronic mail: ganek@us.ibm.com).* Mr. Ganek is Vice President, Autonomic Computing, IBM Software Group. In this recently created role, he leads the IBM corporate-wide initiative for autonomic computing that focuses on making computing systems more self-managing and resilient, lowering the cost of ownership and removing obstacles to growth and flexibility. Prior to joining the Software Group, he was responsible for the technical strategy and operations of the IBM Research Division. Mr. Ganek joined IBM as a software engineer in 1978 in Poughkeepsie, New York, where he was involved in operating system design and development, computer addressing architecture, and parallel systems architecture and design. He was the recipient of IBM Outstanding Innovation Awards for his work on Enterprise Systems Architecture/370™ and System/390® Parallel Sysplex® Design. In 1985, he was appointed manager of MVS™ Design and Performance Analysis and was responsible for the technical plan and content of the MVS control program. Subsequently he was appointed VM/XA advanced system manager, responsible for strat-

egy, design, planning, customer support, system evaluation, and product delivery and control. Mr. Ganek became manager of Enterprise Systems Market Operations in 1989, responsible for System/390 software requirements and announcement strategy. He was appointed Director of Worldwide Software Manufacturing Strategy in 1990, responsible for IBM's strategy for manufacturing, distribution, and packaging of software, software service, and publications across all hardware platforms. In 1992, he was named Programming Systems Director, Quality and Development Operations, leading quality programs for the Programming Systems Division. He joined the Telecommunications and Media Industry Unit in 1994 as Director of Solutions Development, IBM Telecommunications and Media Industry Unit. Mr. Ganek received his M.S. in computer science from Rutgers University in 1981. He holds 15 patents.

**Thomas A. Corbi** *IBM Software Group, Hawthorne II, 17 Sky Line Drive, Hawthorne, New York 10532 (electronic mail: groklib@us.ibm.com).* Mr. Corbi is a member of the Autonomic Computing Architecture team that is working toward creating a framework and standards to enable development of autonomic software systems. He also manages the Autonomic Computing Technology Institute, which is a joint research program between the Software Group and the IBM Research Division. Prior to joining the Software Group, he was a research staff member and technical assistant to the Vice President of Emerging Business Opportunities in IBM Research. He joined IBM at the former Palo Alto Development Center as a programmer in 1974 after graduating from Yale University with a degree in computer science. He worked in the former General Products Division on future systems text-processing, data management, and database systems, and was development manager for IMS™/VS Fast Path at the IBM Santa Teresa Laboratory. Mr. Corbi joined the Research Division at the Thomas J. Watson Research Center in 1982 and managed the Program Understanding project from 1986 to 1988. He cochaired the 1988 IEEE Conference on Software Maintenance (CSM-88). From 1988 to 1990, he went on assignment as technical assistant to the S/390 Operating Systems Division Director in Poughkeepsie, New York, where he wrote "Understanding and Improving Software Development Productivity." He returned to Research and was senior manager of speech recognition development in the Experimental Software Development Center. In 1994, he was assigned to the Research Headquarters Technical Staff, assisting the Vice President for Technical Plans and Controls. Afterwards, Mr. Corbi contributed to a number of research projects, including low power exploratory systems, home networking and consumer systems, pervasive computing systems solutions, and wearable computing platforms. He holds two patents.

# The Vision of Autonomic Computing




**Systems manage themselves according to an administrator's goals. New components integrate as effortlessly as a new cell establishes itself in the human body. These ideas are not science fiction, but elements of the grand challenge to create self-managing computing systems.**

*Jeffrey O. Kephart*
*David M. Chess*
IBM Thomas J. Watson Research Center

In mid-October 2001, IBM released a manifesto observing that the main obstacle to further progress in the IT industry is a looming software complexity crisis.[1] The company cited applications and environments that weigh in at tens of millions of lines of code and require skilled IT professionals to install, configure, tune, and maintain.

The manifesto pointed out that the difficulty of managing today's computing systems goes well beyond the administration of individual software environments. The need to integrate several heterogeneous environments into corporate-wide computing systems, and to extend that beyond company boundaries into the Internet, introduces new levels of complexity. Computing systems' complexity appears to be approaching the limits of human capability, yet the march toward increased interconnectivity and integration rushes ahead unabated.

This march could turn the dream of pervasive computing—trillions of computing devices connected to the Internet—into a nightmare. Programming language innovations have extended the size and complexity of systems that architects can design, but relying solely on further innovations in programming methods will not get us through the present complexity crisis.

As systems become more interconnected and diverse, architects are less able to anticipate and design interactions among components, leaving such issues to be dealt with at runtime. Soon systems will become too massive and complex for even the most skilled system integrators to install, configure, optimize, maintain, and merge. And there will be no way to make timely, decisive responses to the rapid stream of changing and conflicting demands.

## AUTONOMIC OPTION

The only option remaining is *autonomic computing*—computing systems that can manage themselves given high-level objectives from administrators. When IBM's senior vice president of research, Paul Horn, introduced this idea to the National Academy of Engineers at Harvard University in a March 2001 keynote address, he deliberately chose a term with a biological connotation. The autonomic nervous system governs our heart rate and body temperature, thus freeing our conscious brain from the burden of dealing with these and many other low-level, yet vital, functions.

The term autonomic computing is emblematic of a vast and somewhat tangled hierarchy of natural self-governing systems, many of which consist of myriad interacting, self-governing components that in turn comprise large numbers of interacting, autonomous, self-governing components at the next level down. The enormous range in scale, starting with molecular machines within cells and extending to human markets, societies, and the entire world socioeconomy, mirrors that of computing systems, which run from individual devices to the entire Internet. Thus, we believe it will be profitable to seek inspiration in the self-governance of social and economic systems as well as purely biological ones.

Clearly then, autonomic computing is a grand

0018-9162/03/$17.00 © 2003 IEEE

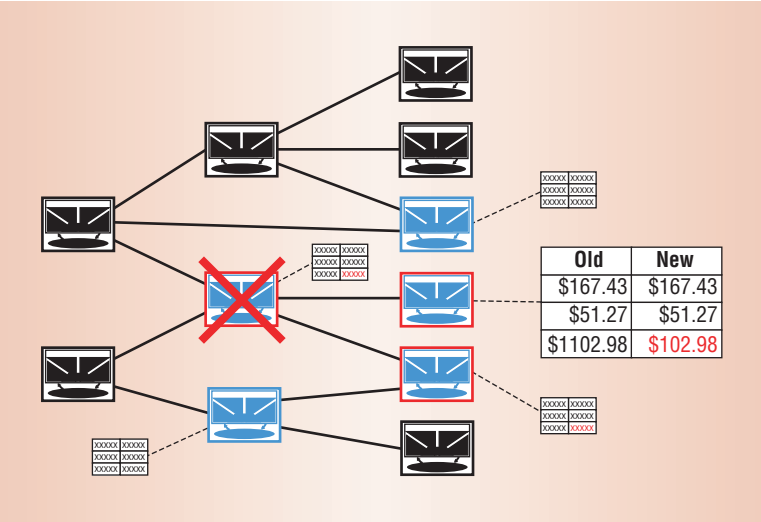



***Figure 1. Problem diagnosis in an autonomic system upgrade. The upgrade introduces five software modules (blue), each an autonomic element. Minutes after installation, regression testers find faulty output in three of the new modules (red outlines), and the system immediately reverts to its old version. A problem determiner, an autonomic element, obtains information about interelement dependencies (lines between elements) from a dependency analyzer, another autonomic element that probes the system periodically (not shown). Taking into account its knowledge of interelement dependencies, the problem determiner analyzes log files and infers which of the three potentially bad modules is the culprit (red X). It generates a problem ticket containing diagnostic information and sends it to a software developer, who debugs the module and makes it available for future upgrades.***

challenge that reaches far beyond a single organization. Its realization will take a concerted, long-term, worldwide effort by researchers in a diversity of fields. A necessary first step is to examine this vision: what autonomic computing systems might look like, how they might function, and what obstacles researchers will face in designing them and understanding their behavior.

## SELF-MANAGEMENT

The essence of autonomic computing systems is self-management, the intent of which is to free system administrators from the details of system operation and maintenance and to provide users with a machine that runs at peak performance 24/7.

Like their biological namesakes, autonomic systems will maintain and adjust their operation in the face of changing components, workloads, demands, and external conditions and in the face of hardware or software failures, both innocent and malicious. The autonomic system might continually monitor its own use, and check for component upgrades, for example. If it deems the advertised features of the upgrades worthwhile, the system will install them, reconfigure itself as necessary, and run a regression test to make sure all is well. When it detects errors, the system will revert to the older version while its automatic problem-determination algorithms try to isolate the source of the error. Figure 1 illustrates how this process might work for an autonomic accounting system upgrade.

IBM frequently cites four aspects of self-management, which Table 1 summarizes. Early autonomic systems may treat these aspects as distinct, with different product teams creating solutions that address each one separately. Ultimately, these aspects will be emergent properties of a general architecture, and distinctions will blur into a more general notion of self-maintenance.

The journey toward fully autonomic computing will take many years, but there are several important and valuable milestones along the path. At first, automated functions will merely collect and aggregate information to support decisions by human administrators. Later, they will serve as advisors, suggesting possible courses of action for humans to consider. As automation technologies improve, and our faith in them grows, we will entrust autonomic systems with making—and acting on—lower-level decisions. Over time, humans will need to make relatively less frequent predominantly higher-level decisions, which the system will carry out automatically via more numerous, lower-level decisions and actions.

Ultimately, system administrators and end users will take the benefits of autonomic computing for granted. Self-managing systems and devices will seem completely natural and unremarkable, as will automated software and middleware upgrades. The detailed migration patterns of applications or data will be as uninteresting to us as the details of routing a phone call through the telephone network.

### Self-configuration

Installing, configuring, and integrating large, complex systems is challenging, time-consuming, and error-prone even for experts. Most large Web sites and corporate data centers are haphazard accretions of servers, routers, databases, and other technologies on different platforms from different vendors. It can take teams of expert programmers months to merge two systems or to install a major e-commerce application such as SAP.

Autonomic systems will configure themselves automatically in accordance with high-level policies—representing business-level objectives, for

**Table 1. Four aspects of self-management as they are now and would be with autonomic computing.**

| Concept | Current computing | Autonomic computing |
|---|---|---|
| Self-configuration | Corporate data centers have multiple vendors and platforms. Installing, configuring, and integrating systems is time consuming and error prone. | Automated configuration of components and systems follows high-level policies. Rest of system adjusts automatically and seamlessly. |
| Self-optimization | Systems have hundreds of manually set, nonlinear tuning parameters, and their number increases with each release. | Components and systems continually seek opportunities to improve their own performance and efficiency. |
| Self-healing | Problem determination in large, complex systems can take a team of programmers weeks. | System automatically detects, diagnoses, and repairs localized software and hardware problems. |
| Self-protection | Detection of and recovery from attacks and cascading failures is manual. | System automatically defends against malicious attacks or cascading failures. It uses early warning to anticipate and prevent systemwide failures. |

example—that specify what is desired, not how it is to be accomplished. When a component is introduced, it will incorporate itself seamlessly, and the rest of the system will adapt to its presence—much like a new cell in the body or a new person in a population. For example, when a new component is introduced into an autonomic accounting system, as in Figure 1, it will automatically learn about and take into account the composition and configuration of the system. It will register itself and its capabilities so that other components can either use it or modify their own behavior appropriately.

### Self-optimization

Complex middleware, such as WebSphere, or database systems, such as Oracle or DB2, may have hundreds of tunable parameters that must be set correctly for the system to perform optimally, yet few people know how to tune them. Such systems are often integrated with other, equally complex systems. Consequently, performance-tuning one large subsystem can have unanticipated effects on the entire system.

Autonomic systems will continually seek ways to improve their operation, identifying and seizing opportunities to make themselves more efficient in performance or cost. Just as muscles become stronger through exercise, and the brain modifies its circuitry during learning, autonomic systems will monitor, experiment with, and tune their own parameters and will learn to make appropriate choices about keeping functions or outsourcing them. They will proactively seek to upgrade their function by finding, verifying, and applying the latest updates.

### Self-healing

IBM and other IT vendors have large departments devoted to identifying, tracing, and determining the root cause of failures in complex computing systems. Serious customer problems can take teams of programmers several weeks to diagnose and fix, and sometimes the problem disappears mysteriously without any satisfactory diagnosis.

Autonomic computing systems will detect, diagnose, and repair localized problems resulting from bugs or failures in software and hardware, perhaps through a regression tester, as in Figure 1. Using knowledge about the system configuration, a problem-diagnosis component (based on a Bayesian network, for example) would analyze information from log files, possibly supplemented with data from additional monitors that it has requested. The system would then match the diagnosis against known software patches (or alert a human programmer if there are none), install the appropriate patch, and retest.

### Self-protection

Despite the existence of firewalls and intrusion-detection tools, humans must at present decide how to protect systems from malicious attacks and inadvertent cascading failures.

Autonomic systems will be self-protecting in two senses. They will defend the system as a whole against large-scale, correlated problems arising from malicious attacks or cascading failures that remain uncorrected by self-healing measures. They also will anticipate problems based on early reports from sensors and take steps to avoid or mitigate them.

## ARCHITECTURAL CONSIDERATIONS

Autonomic systems will be interactive collections of *autonomic elements*—individual system constituents that contain resources and deliver services to humans and other autonomic elements. Autonomic elements will manage their internal behavior and their relationships with other autonomic elements in accordance with policies that humans or other elements have established. *System* self-management will arise at least as much from the myriad

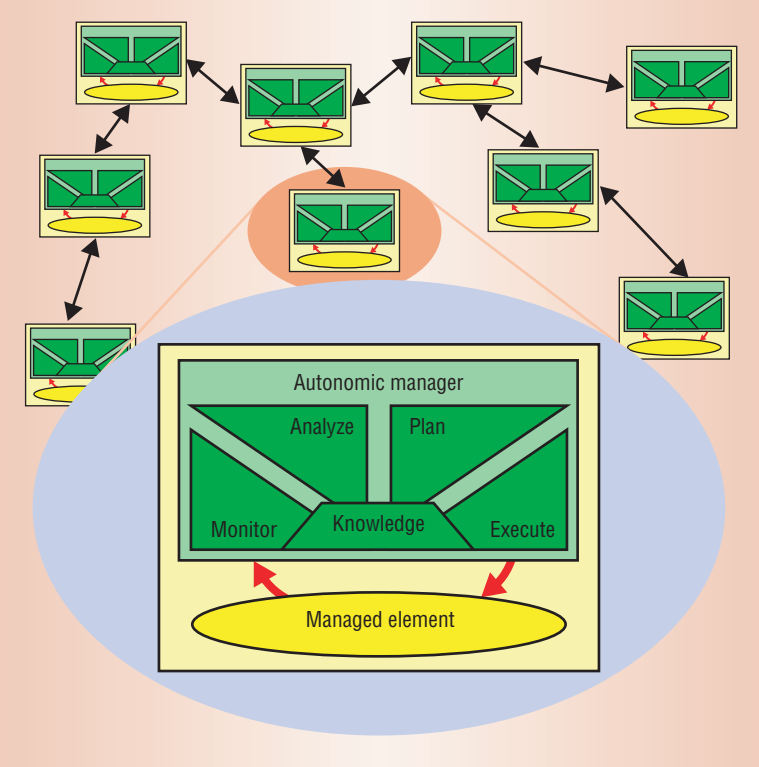



***Figure 2. Structure of an autonomic element. Elements interact with other elements and with human programmers via their autonomic managers.***

interactions among autonomic elements as it will from the internal self-management of the individual autonomic elements—just as the social intelligence of an ant colony arises largely from the interactions among individual ants. A distributed, service-oriented infrastructure will support autonomic elements and their interactions.

As Figure 2 shows, an autonomic element will typically consist of one or more managed elements coupled with a single autonomic manager that controls and represents them. The managed element will essentially be equivalent to what is found in ordinary nonautonomic systems, although it can be adapted to enable the autonomic manager to monitor and control it. The managed element could be a hardware resource, such as storage, a CPU, or a printer, or a software resource, such as a database, a directory service, or a large legacy system.

At the highest level, the managed element could be an e-utility, an application service, or even an individual business. The autonomic manager distinguishes the autonomic element from its nonautonomic counterpart. By monitoring the managed element and its external environment, and constructing and executing plans based on an analysis of this information, the autonomic manager will relieve humans of the responsibility of directly managing the managed element.

Fully autonomic computing is likely to evolve as designers gradually add increasingly sophisticated autonomic managers to existing managed elements. Ultimately, the distinction between the autonomic manager and the managed element may become merely conceptual rather than architectural, or it may melt away—leaving fully integrated, autonomic elements with well-defined behaviors and interfaces, but also with few constraints on their internal structure.

Each autonomic element will be responsible for managing its own internal state and behavior and for managing its interactions with an environment that consists largely of signals and messages from other elements and the external world. An element's internal behavior and its relationships with other elements will be driven by goals that its designer has embedded in it, by other elements that have authority over it, or by subcontracts to peer elements with its tacit or explicit consent. The element may require assistance from other elements to achieve its goals. If so, it will be responsible for obtaining necessary resources from other elements and for dealing with exception cases, such as the failure of a required resource.

Autonomic elements will function at many levels, from individual computing components such as disk drives to small-scale computing systems such as workstations or servers to entire automated enterprises in the largest autonomic system of all—the global economy.

At the lower levels, an autonomic element's range of internal behaviors and relationships with other elements, and the set of elements with which it can interact, may be relatively limited and hard-coded. Particularly at the level of individual components, well-established techniques—many of which fall under the rubric of fault tolerance—have led to the development of elements that rarely fail, which is one important aspect of being autonomic. Decades of developing fault-tolerance techniques have produced such engineering feats as the IBM zSeries servers, which have a mean time to failure of several decades.

At the higher levels, fixed behaviors, connections, and relationships will give way to increased dynamism and flexibility. All these aspects of autonomic elements will be expressed in more high-level, goal-oriented terms, leaving the elements themselves with the responsibility for resolving the details on the fly.

Hard-coded behaviors will give way to behaviors expressed as high-level objectives, such as "maximize this utility function," or "find a reputable message translation service." Hardwired connections among elements will give way to increasingly less direct specifications of an element's partners—from specification by physical address to specification by name and finally to specification by function, with the partner's identity being resolved only when it is needed.

Hard-wired relationships will evolve into flexible relationships that are established via negotiation. Elements will automatically handle new modes of failure, such as contract violation by a supplier, without human intervention.

While service-oriented architectural concepts like Web and grid services[2,3] will play a fundamental role, a sufficient foundation for autonomic computing requires more. First, as service providers, autonomic elements will not unquestioningly honor requests for service, as would typical Web services or objects in an object-oriented environment. They will provide a service only if providing it is consistent with their goals. Second, as consumers, autonomic elements will autonomously and proactively issue requests to other elements to carry out their objectives.

Finally, autonomic elements will have complex life cycles, continually carrying on multiple threads of activity, and continually sensing and responding to the environment in which they are situated. Autonomy, proactivity, and goal-directed interactivity with their environment are distinguishing characteristics of software agents. Viewing autonomic elements as agents and autonomic systems as multiagent systems makes it clear that agent-oriented architectural concepts will be critically important.[4]

## ENGINEERING CHALLENGES

Virtually every aspect of autonomic computing offers significant engineering challenges. The life cycle of an individual autonomic element or of a relationship among autonomic elements reveals several challenges. Others arise in the context of the system as a whole, and still more become apparent at the interface between humans and autonomic systems.

### Life cycle of an autonomic element

An autonomic element's life cycle begins with its design and implementation; continues with test and verification; proceeds to installation, configuration, optimization, upgrading, monitoring, problem determination, and recovery; and culminates in uninstallation or replacement. Each of these stages has special issues and challenges.

**Design, test, and verification.** Programming an autonomic element will mean extending Web services or grid services with programming tools and techniques that aid in managing relationships with other autonomic elements. Because autonomic elements both consume and provide services, representing needs and preferences will be just as important as representing capabilities. Programmers will need tools that help them acquire and represent policies—high-level specifications of goals and constraints, typically represented as rules or utility functions—and map them onto lower-level actions. They will also need tools to build elements that can establish, monitor, and enforce agreements.

**Autonomic elements will provide a service only if it is consistent with their goals.**

Testing autonomic elements and verifying that they behave correctly will be particularly challenging in large-scale systems because it will be harder to anticipate their environment, especially when it extends across multiple administrative domains or enterprises. Testing networked applications that require coordinated interactions among several autonomic elements will be even more difficult.

It will be virtually impossible to build test systems that capture the size and complexity of realistic systems and workloads. It might be possible to test newly deployed autonomic elements in situ by having them perform alongside more established and trusted elements with similar functionality.

The element's potential customers may also want to test and verify its behavior, both before establishing a service agreement and while the service is provided. One approach is for the autonomic element to attach a testing method to its service description.

**Installation and configuration.** Installing and configuring autonomic elements will most likely entail a bootstrapping process that begins when the element registers itself in a directory service by publishing its capabilities and contact information. The element might also use the directory service to discover suppliers or brokers that may provide information or services it needs to complete its initial configuration. It can also use the service to seek out potential customers or brokers to which it can delegate the task of finding customers.

**Monitoring and problem determination.** Monitoring will be an essential feature of autonomic elements. Elements will continually monitor themselves to ensure that they are meeting their own objectives, and they will log this information to serve as the

**The vision of autonomic systems as a complex supply web makes problem determination both easier and harder than it is now.**

basis for adaptation, self-optimization, and reconfiguration. They will also continually monitor their suppliers to ensure that they are receiving the agreed-on level of service and their customers to ensure that they are not exceeding the agreed-on level of demand. Special sentinel elements may monitor other elements and issue alerts to interested parties when they fail.

When coupled with event correlation and other forms of analysis, monitoring will be important in supporting problem determination and recovery when a fault is found or suspected. Applying monitoring, audit, and verification tests at all the needed points without burdening systems with excessive bandwidth or processing demands will be a challenge. Technologies to allow statistical or sample-based testing in a dynamic environment may prove helpful.

The vision of autonomic systems as a complex supply web makes problem determination both easier and harder than it is now. An autonomic element that detects poor performance or failure in a supplier may not attempt a diagnosis; it may simply work around the problem by finding a new supplier.

In other situations, however, it will be necessary to determine why one or more elements are failing, preferably without shutting down and restarting the entire system. This requires theoretically grounded tools for tracing, simulation, and problem determination in complex dynamic environments. Particularly when autonomic elements—or applications based on interactions among multiple elements—have a large amount of state, recovering gracefully and quickly from failure or restarting applications after software has been upgraded or after a function has been relocated to new machines will be challenging. David Patterson and colleagues at the University of California, Berkeley, and Stanford University have made a promising start in this direction.[5]

**Upgrading.** Autonomic elements will need to upgrade themselves from time to time. They might subscribe to a service that alerts them to the availability of relevant upgrades and decide for themselves when to apply the upgrade, possibly with guidance from another element or a human. Alternatively, the system could create afresh entirely new elements as part of a system upgrade, eliminating outmoded elements only after the new ones establish that they are working properly.

**Managing the life cycle.** Autonomic elements will typically be engaged in many activities simultaneously: participating in one or more negotiations at various phases of completion, proactively seeking inputs from other elements, and so on. They will need to schedule and prioritize their myriad activities, and they will need to represent their life cycle so that they can both reason about it and communicate it to other elements.

## Relationships among autonomic elements

In its most dynamic and elaborate form, the service relationship among autonomic elements will also have a life cycle. Each stage of this life cycle engenders its own set of engineering challenges and standardization requirements.

**Specification.** An autonomic element must have associated with it a set of output services it can perform and a set of input services that it requires, expressed in a standard format so that other autonomic elements can understand it. Typically, the element will register with a directory service such as Universal Description, Discovery, and Integration[6] or an Open Grid Services Architecture (OGSA) registry,[3] providing a description of its capabilities and details about addresses and the protocols other elements or people can use to communicate with it.

Establishing standard service ontologies and a standard service description syntax and semantics that are sufficiently expressive for machines to interpret and reason about is an area of active research. The US Defense Advanced Research Projects Agency's semantic Web effort[7] is representative.

**Location.** An autonomic element must be able to locate input services that it needs; in turn, other elements that require its output services must be able to locate that element.

To locate other elements dynamically, the element can look them up by name or function in a directory service, possibly using a search process that involves sophisticated reasoning about service ontologies. The element can then contact one or more potential service providers directly and converse with them to determine if it can provide exactly the service they require.

In many cases, autonomic elements will also need to judge the likely reliability or trustworthiness of potential partners—an area of active research with many unsolved fundamental problems.

**Negotiation.** Once an element finds potential providers of an input service, it must negotiate with them to obtain that service.

We construe negotiation broadly as any process by which an agreement is reached. In *demand-for-service* negotiation, the element providing a service is subservient to the one requesting it, and the

provider must furnish the service unless it does not have sufficient resources to do so. Another simple form of negotiation is *first-come, first-served*, in which the provider satisfies all requests until it runs into resource limitations. In *posted-price* negotiation, the provider sets a price in real or artificial currency for its service, and the requester must take it or leave it.

More complex forms of negotiation include bilateral or multilateral negotiations over multiple attributes, such as price, service level, and priority, involving multiple rounds of proposals and counterproposals. A third-party arbiter can run an auction or otherwise assist these more complex negotiations, especially when they are multilateral.

Negotiation will be a rich source of engineering and scientific challenges for autonomic computing. Elements need flexible ways to express multiattribute needs and capabilities, and they need mechanisms for deriving these expressions from human input or from computation. They also need effective negotiation strategies and protocols that establish the rules of negotiation and govern the flow of messages among the negotiators. There must be languages for expressing service agreements—the culmination of successful negotiation—in their transient and final forms.

Efforts to standardize the representation of agreements are under way, but mechanisms for negotiating, enforcing, and reasoning about agreements are lacking, as are methods for translating them into action plans.

**Provision.** Once two elements reach an agreement, they must provision their internal resources. Provision may be as simple as noting in an access list that a particular element can request service in the future, or it may entail establishing additional relationships with other elements, which become subcontractors in providing some part of the agreed-on service or task.

**Operation.** Once both sides are properly provisioned, they operate under the negotiated agreement. The service provider's autonomic manager oversees the operation of its managed element, monitoring it to ensure that the agreement is being honored; the service requester might similarly monitor the level of service.

If the agreement is violated, one or both elements would seek an appropriate remedy. The remedy may be to assess a penalty, renegotiate the agreement, take technical measures to minimize any harm from the failure, or even terminate the agreement.

**Termination.** When the agreement has run its course, the parties agree to terminate it, freeing their internal resources for other uses and terminating agreements for input services that are no longer needed. The parties may record pertinent information about the service relationship locally, or store it in a database a reputation element maintains.

## Systemwide issues

Other important engineering issues that arise at the system level include security, privacy, and trust, and the emergence of new types of services to serve the needs of other autonomic elements.

**System-level engineering issues include security, privacy, and trust, and new types of services to serve the needs of other autonomic elements.**

Autonomic computing systems will be subject to all the security, privacy, and trust issues that traditional computing systems must now address. Autonomic elements and systems will need to both establish and abide by security policies, just as human administrators do today, and they will need to do so in an understandable and fail-safe manner.

Systems that span multiple administrative domains—especially those that cross company boundaries—will face many of the challenges that now confront electronic commerce. These include authentication, authorization, encryption, signing, secure auditing and monitoring, nonrepudiation, data aggregation and identity masking, and compliance with complex legal requirements that vary from state to state or country to country.

The autonomic systems infrastructure must let autonomic elements identify themselves, verify the identities of other entities with which they communicate, verify that a message has not been altered in transit, and ensure that unauthorized parties do not read messages and other data. To satisfy privacy policies and laws, elements must also appropriately protect private and personal information that comes into their possession. Measures that keep data segregated according to its origin or its purpose must be extended into the realm of autonomic elements to satisfy policy and legal requirements.

Autonomic systems must be robust against new and insidious forms of attack that use self-management based on high-level policies to their own advantage. By altering or otherwise manipulating high-level policies, an attacker could gain much greater leverage than is possible in nonautonomic systems. Preventing such problems may require a new subfield of computer security that seeks to thwart fraud and the fraudulent persuasion of autonomic elements.

On a larger scale, autonomic elements will be

**To satisfy privacy policies and laws, elements must appropriately protect information that comes into their possession.**

agents, and autonomic systems will in effect be multiagent systems built on a Web services or OGSA infrastructure. Autonomic systems will be inhabited by middle agents[8] that serve as intermediaries of various types, including directory services, matchmakers, brokers, auctioneers, data aggregators, dependency managers—for detecting, recording, and publicizing information about functional dependencies among autonomic elements—event correlators, security analysts, time-stampers, sentinels, and other types of monitors that assess the health of other elements or of the system as a whole.

Traditionally, many of these services have been part of the system infrastructure; in a multiagent, autonomic world, moving them out of the infrastructure and representing them as autonomic elements themselves will be more natural and flexible.

### Goal specification

While autonomic systems will assume much of the burden of system operation and integration, it will still be up to humans to provide those systems with policies—the goals and constraints that govern their actions. The enormous leverage of autonomic systems will greatly reduce human errors, but it will also greatly magnify the consequences of any error humans do make in specifying goals.

The indirect effect of policies on system configuration and behavior exacerbates the problem because tracing and correcting policy errors will be very difficult. It is thus critical to ensure that the specified goals represent what is really desired. Two engineering challenges stem from this mandate: Ensure that goals are specified correctly in the first place, and ensure that systems behave reasonably even when they are not.

In many cases, the set of goals to be specified will be complex, multidimensional, and conflicting. Even a goal as superficially simple as "maximize utility" will require a human to express a complicated multiattribute utility function. A key to reducing error will be to simplify and clarify the means by which humans express their goals to computers. Psychologists and computer scientists will need to work together to strike the right balance between overwhelming humans with too many questions or too much information and underempowering them with too few options or too little information.

The second challenge—ensuring reasonable system behavior in the face of erroneous input—is another facet of robustness: Autonomic systems will need to protect themselves from input goals that are inconsistent, implausible, dangerous, or unrealizable with the resources at hand. Autonomic systems will subject such inputs to extra validation, and when self-protective measures fail, they will rely on deep-seated notions of what constitutes acceptable behavior to detect and correct problems. In some cases, such as resource overload, they will inform human operators about the nature of the problem and offer alternative solutions.

## SCIENTIFIC CHALLENGES

The success of autonomic computing will hinge on the extent to which theorists can identify universal principles that span the multiple levels at which autonomic systems can exist—from systems to enterprises to economies.

### Behavioral abstractions and models

Defining appropriate abstractions and models for understanding, controlling, and designing emergent behavior in autonomic systems is a challenge at the heart of autonomic computing. We need fundamental mathematical work aimed at understanding how the properties of self-configuration, self-optimization, self-maintenance, and robustness arise from or depend on the behaviors, goals, and adaptivity of individual autonomic elements; the pattern and type of interactions among them; and the external influences or demands on the system.

Understanding the mapping from local behavior to global behavior is a necessary but insufficient condition for controlling and designing autonomic systems. We must also discover how to exploit the inverse relationship: How can we derive a set of behavioral and interaction rules that, if embedded in individual autonomic elements, will induce a desired global behavior? The nonlinearity of emergent behavior makes such an inversion highly nontrivial.

One plausible approach couples advanced search and optimization techniques with parameterized models of the local-to-global relationship and the likely set of environmental influences to which the system will be subjected. Melanie Mitchell and colleagues[9] at the Santa Fe Institute have pioneered this approach, using genetic algorithms to evolve the local transformation rules of simple cellular automata to achieve desired global behaviors. At NASA, David Wolpert and colleagues[10] have studied algorithms that, given a high-level global objective, derive individual goals for individual agents. When each agent selfishly follows its goals, the desired global behavior results.

These methods are just a start. We have yet to understand fundamental limits on what classes of

global behavior can be achieved, nor do we have practical methods for designing emergent system behavior. Moreover, although these methods establish the rules of a system at design time, autonomic systems must deal with shifting conditions that can be known only at runtime. Control theoretic approaches may prove useful in this capacity; some autonomic managers may use control systems to govern the behavior of their associated managed elements.

The greatest value may be in extending distributed or hierarchical control theories, which consider interactions among independently or hierarchically controlled elements, rather than focusing on an individual controlled element. Newer paradigms for control may be needed when there is no clear separation of scope or time scale.

### Robustness theory

A related challenge is to develop a theory of robustness for autonomic systems, including definitions and analyses of robustness, diversity, redundancy, and optimality and their relationship to one another. The Santa Fe Institute recently began a multidisciplinary study on this topic (http://discuss.santafe.edu/robustness).

### Learning and optimization theory

Machine learning by a single agent in relatively static environments is well studied, and it is well supported by strong theoretical results. However, in more sophisticated autonomic systems, individual elements will be agents that continually adapt to their environment—an environment that consists largely of other agents. Thus, even with stable external conditions, agents are adapting to one another, which violates the traditional assumptions on which single-agent learning theories are based.

There are no guarantees of convergence. In fact, interesting forms of instability have been observed in such cases.[11] Learning in multiagent systems is a challenging but relatively unexplored problem, with virtually no major theorems and only a handful of empirical results.

Just as learning becomes a more challenging problem in multiagent systems, so does optimization. The root cause is the same—whether it is because they are learning or because they are optimizing, agents are changing their behavior, making it necessary for other agents to change their behavior, potentially leading to instabilities. Optimization in such an environment must deal with dynamics created by a collective mode of oscillation rather than a drifting environmental signal. Optimization techniques that assume a stationary environment have been observed to fail pathologically in multiagent systems,[12] therefore they must either be revamped or replaced with new methods.

**Optimization techniques that assume a stationary environment must either be revamped or replaced with new methods.**

### Negotiation theory

A solid theoretical foundation for negotiation must take into account two perspectives. From the perspective of individual elements, we must develop and analyze algorithms and negotiation protocols and determine what bidding or negotiation algorithms are most effective.

From the perspective of the system as a whole, we must establish how overall system behavior depends on the mixture of negotiation algorithms that various autonomic elements use and establish the conditions under which multilateral—as opposed to bilateral—negotiations among elements are necessary or desirable.

### Automated statistical modeling

Statistical models of large networked systems will let autonomic elements or systems detect or predict overall performance problems from a stream of sensor data from individual devices. At long time scales—during which the configuration of the system changes—we seek methods that automate the aggregation of statistical variables to reduce the dimensionality of the problem to a size that is amenable to adaptive learning and optimization techniques that operate on shorter time scales.

Is it possible to meet the grand challenge of autonomic computing without magic and without fully solving the AI problem? We believe it is, but it will take time and patience. Long before we solve many of the more challenging problems, less automated realizations of autonomic systems will be extremely valuable, and their value will increase substantially as autonomic computing technology improves and earns greater trust and acceptance.

A vision this large requires that we pool expertise in many areas of computer science as well as in disciplines that lie far beyond computing's traditional boundaries. We must look to scientists studying nonlinear dynamics and complexity for new theories of emergent phenomena and robustness. We must look to economists and e-commerce researchers for ideas and technologies about negotiation and supply webs. We must look to psychologists and human factors researchers for new goal-definition and visualization paradigms and

for ways to help humans build trust in autonomic systems. We must look to the legal profession, since many of the same issues that arise in the context of e-commerce will be important in autonomic systems that span organizational or national boundaries.

Bridging the language and cultural divides among the many disciplines needed for this endeavor and harnessing the diversity to yield successful and perhaps universal approaches to autonomic computing will perhaps be the greatest challenge. It will be interesting to see what new cross-disciplines develop as we begin to work together to solve these fundamental problems. ■


## Acknowledgments

We are indebted to the many people who influenced this article with their ideas and thoughtful criticisms. Special thanks go to David Chambliss for contributing valuable thoughts on human-computer interface issues. We also thank Bill Arnold, David Bantz, Rob Barrett, Peter Capek, Alan Ganek, German Goldszmidt, James Hanson, Joseph Hellerstein, James Kozloski, Herb Lee, Charles Peck, Ed Snible, and Ian Whalley for their helpful comments, and the members of an IBM Academy of Technology team for their extensive written and verbal contributions: Lisa Spainhower and Kazuo Iwano (co-leaders), William H. Tetzlaff (Technology Council contact), Robert Abrams, Sam Adams, Steve Burbeck, Bill Chung, Denise Y. Dyko, Stuart Feldman, Lorraine Herger, Mark Johnson, James Kaufman, David Kra, Ed Lassettre, Andreas Maier, Timothy Marchini, Norm Pass, Colin Powell, Stephen A. Smithers, Daniel Sturman, Mark N. Wegman, Steve R. White, and Daniel Yellin.


## References


1. IBM, "Autonomic Computing: IBM's Perspective on the State of Information Technology"; http://www-1.ibm.com/industries/government/doc/content/resource/thought/278606109.html.
2. H. Kreger, "Web Services Conceptual Architecture," v. 1.0. 2001; http://www-4.ibm.com/software/solutions/webservices/pdf/WSCA.pdf.
3. I. Foster et al., "The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration," Feb. 2002; http://www.globus.org/research/papers/ogsa.pdf.
4. N.R. Jennings, "On Agent-Based Software Engineering," *Artificial Intelligence*, vol. 177, no. 2, 2000, pp. 277-296.
5. D. Patterson et al., *Recovery-Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies*, tech. report CSD-02-1175, Computer Science Dept., Univ. of Calif., Berkeley, Calif., Mar. 2002.
6. Ariba, IBM, and Microsoft, "UDDI Technical White Paper," 2000; http://www.uddi.org/whitepapers.html.
7. T. Berners-Lee, J. Hendler, and O. Lassila, "The Semantic Web," *Scientific American*, May 2001, pp. 28-37.
8. H. Wong and K. Sycara, "A Taxonomy of Middle Agents for the Internet," *Proc. 4th Int'l Conf. Multiagent Systems*, IEEE CS Press, 2000, pp. 465-466.
9. R. Das et al., "Evolving Globally Synchronized Cellular Automata," *Proc. 6th Int'l Conf. Genetic Algorithms*, L. Eshelman, ed., Morgan Kaufmann, 1995, pp. 336-343.
10. D. Wolpert, K. Wheeler, and K. Tumer, *Collective Intelligence for Control of Distributed Dynamical Systems*, tech. report NASA-ARC-IC-99-44, NASA, Ames, Iowa, 1999.
11. J.O. Kephart and G.J. Tesauro, "Pseudo-Convergent Q-Learning by Competitive Pricebots," *Proc. 17th Int'l Conf. Machine Learning*, Morgan Kaufmann, 2000, pp. 463-470.
12. J.O. Kephart et al., "Pricing Information Bundles in a Dynamic Environment," *Proc. 3rd ACM Conf. Electronic Commerce*, 2001, ACM Press, pp. 180-190.



***Jeffrey O. Kephart*** *manages the Agents and Emergent Phenomena group at the IBM Thomas J. Watson Research Center. His research focuses on the application of analogies from biology and economics to massively distributed computing systems, particularly in the domains of autonomic computing, e-commerce, and antivirus technology. Kephart received a BS from Princeton University and a PhD from Stanford University, both in electrical engineering. Contact him at kephart@us.ibm.com.*

***David M. Chess*** *is a research staff member at the IBM Thomas J. Watson Research Center, working in autonomic computing and computer security. He received a BA in philosophy from Princeton University and an MS in computer science from Pace University. Contact him at chess@us.ibm.com.*

*Tivoli software: Intelligent management software that integrates and automates*

Tivoli® software

IBM

# The Tivoli software implementation of autonomic computing guidelines

## Introducing autonomic computing

Autonomic computing is the ability of IT infrastructures to adapt to change in a complex environment in accordance with business policies and objectives. This brochure defines the concepts and value of autonomic computing; the autonomic self-managing characteristics and its affect on process, tools and skills within the IT organization; and the evolution of systems and technology in autonomic computing.

Autonomic computing systems have the ability to perform management activities based on situations they observe or sense in the IT environment. Rather than IT professionals initiating management activities, the system observes something about itself and acts accordingly—it becomes self-managing.

Autonomic computing is the self-management of e-business infrastructure, balancing what the IT professional manages and what the system manages. It is the evolution of e-business.

*Autonomic computing helps enable enterprises to operate efficiently while decreasing costs and enhancing your company's ability to react to change. Users are freed from mundane tasks and can refocus on defining your company's business priorities.*

**Why autonomic computing?**

Why is autonomic computing important today? The cost of technology continues to decrease yet overall IT costs do not. With the expense challenges that many companies face, IT managers are looking for ways to improve the return on investment of IT by reducing total cost of ownership, improving quality of service, accelerating time to value and managing IT complexity. e-business is a reality, and outages are proving to be expensive and embarrassing.

Adding to this complexity is the proliferation of heterogeneous vendor and technology environments, which require the components of a given solution to be integrated and customized into unique customer business processes. The increased need to distribute data, applications and system resources across geographic and business boundaries further contributes to the complexity of the IT infrastructure.

**What is autonomic computing?**

Autonomic computing is about freeing IT professionals to focus on high-value tasks by making technology work smarter. This means letting computing systems and infrastructure take care of managing themselves. Ultimately, it is writing business policies and goals and letting the infrastructure configure, heal and optimize itself according to those policies while protecting itself from malicious activities.

In an autonomic environment the IT infrastructure and its components are self-configuring, self-healing, self-optimizing and self-protecting. They communicate with each other and with high-level management tools. They regulate themselves and, sometimes, each other in accordance with business priorities. Together they proactively manage the IT environment under the protection of suitable controls while hiding the inherent complexity of these activities from end users.

Collectively these intuitive and collaborative qualities enable enterprises to operate efficiently with fewer human resources, decreasing costs and enhancing a company's ability to react to change. Users are freed from mundane tasks and can focus on defining the business objectives that the autonomic system will use to govern itself.

**How does autonomic computing affect your business?**

No matter how large or how small, IT organizations perform similar sets of tasks to manage their IT environment. Because autonomic computing is about shifting the burden of managing systems from people to technology, it is important to understand these sets of tasks and verify that they are synchronized with autonomic computing.

The business of IT can be described as a collection of practices or processes that organize the work of the IT staff to deliver service. A popular example of this is the IT Information Library (ITIL)—a set of best practices that apply to IT organizations regardless of a company's size or the technology used.

A framework of best-practices disciplines serves as a guide during the shift from you managing your IT infrastructure to the autonomic computing technology handling it. However, it does not happen overnight and cannot be solely accomplished through the acquisition of new products. Skills within the organization need to adapt, and processes need to change, creating new benchmarks of success. By coupling best practices with comprehensive enabling technology, organizations can realize substantial economic value.


**What is the value of autonomic computing?**

Autonomic computing helps address market conditions by using technology to manage technology, helping reduce operational costs and improve return on investment. In autonomic computing, IT infrastructure components take on the following characteristics: self-configuring, self-healing, self-optimizing and self-protecting. IT infrastructure components need to expose manageability characteristics to each other and to high-level management tools.

***Self-configuring***

With the ability to dynamically configure itself on the fly, an IT environment can adapt immediately—with minimal human intervention—to the deployment of new components or changes in the IT environment. Dynamic adaptation helps verify continuous strength and productivity of an e-business infrastructure—often the single-determining factor between business growth and chaos.

***Self-healing***

Self-healing IT environments can detect problematic operations (either proactively through predictions or otherwise) and then initiate corrective action without disrupting system applications. Corrective action could mean that a product alters its own state or influences changes in other elements of the environment. Day-to-day operations do not falter or fail because of events at the component level. The IT environment as a whole becomes more resilient as changes are made to reduce or help eliminate the business impact of failing components.

***Self-optimizing***

Self-optimization refers to the ability of the IT environment to efficiently maximize resource allocation and utilization to meet end users' needs with minimal human intervention. In the near term self-optimization primarily addresses the complexity of managing system performance. In the long term self-optimizing components may learn from experience and automatically and proactively tune themselves in the context of an overall business objective. Self-optimization verifies optimum Quality of Service for both system users and their customers.

***Self-protecting***

A self-protecting environment lets the right people access the right data at the right time and can take appropriate actions automatically to make itself less vulnerable to attacks on its runtime infrastructure and business data.

A self-protecting IT environment can detect hostile or intrusive behavior as it occurs and take autonomous actions to make itself less vulnerable to unauthorized access and use, viruses, denial-of-service attacks and general failures. The self-protecting capability allows businesses to consistently enforce security and privacy policies, reduce overall security administration costs and ultimately increase employee productivity and customer satisfaction.

**The results of autonomic computing**

Below are a few examples of the results delivered by implementing autonomic computing solutions with self-managing characteristics:

***Operational efficiency***

- *As IT infrastructure becomes more autonomic, executing on business policy becomes the focus of IT management. Management of the business and of IT will no longer be separate or possibly conflicting activities.*
- *Self-configuring and self-optimizing technologies drive efficiency in running and deploying new processes and capabilities.*

***Supporting business needs with IT***

- *The actualization of self-configuring systems speeds the deployment of new applications required to support emerging business requirements.*
- *Self-healing capabilities help deliver the 24x7 availability required to keep businesses running.*

***Workforce productivity***

- *Workforce productivity is enhanced when the focus is on management of business processes and policies, without the need to translate these needs into actions that separately manage supporting technology.*
- *Systems that self-manage free up IT resource to move from mundane system management tasks to focus on working with users to solve business problems.*

Delivering systemwide autonomic environments is an evolutionary process that is enabled by technology, but is ultimately implemented by each enterprise through the adoption of these technologies along with supporting processes.

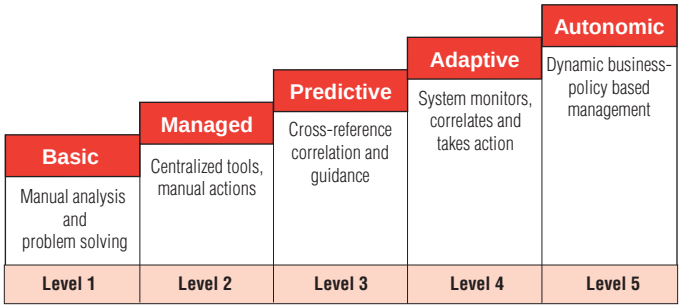
**Levels of autonomic computing: an evolution, not a revolution**

The progression toward an autonomic environment is an evolutionary process enabled by technology but is ultimately implemented by each enterprise through the adoption of these technologies and supporting processes. Each facet of autonomic computing—self-configuring, self-healing, self-optimizing and self-protecting—offers unique challenges and opportunities. Customers can benefit from several common values as they progress from level to level, including:

- *Increase critical system availability*
- *Leverage existing technical and staff resources*
- *Improve responsiveness to change*
- *Improve comprehensive performance*
- *Reduce cycle time for deployment of new systems*

> *"Autonomic computing is not a destination: it's a step-by-step journey which enables long-term value to be sustained from IT investment. IT management technology suppliers like Tivoli software can only deliver on the vision of autonomic computing if they help customers take that journey, one step at a time– by providing best-practices knowledge transfer and technology innovation."*
>
> —Neil Ward-Dutton, Ovum

What's the journey we must take, and what technologies are needed to help get us there? The path to autonomic computing can be thought of in five levels. These levels, defined below, start at basic and continue through managed, predictive, adaptive and finally autonomic.




1. **Basic level**—a starting point of IT environments, where some IT environments are today. Each infrastructure element is managed independently by IT professionals who set it up, monitor it and eventually replace it.
2. **Managed level**—systems management technologies can be used to collect information from disparate systems onto fewer consoles, helping reduce the time it takes for the administrator to collect and synthesize information as the IT environment becomes more complex.
3. **Predictive level**—new technologies are introduced to provide correlation among several infrastructure elements. These elements can begin to recognize patterns, predict the optimal configuration and provide advice on what course of action the administrator should take.
4. **Adaptive level**—technologies improve and people become more comfortable with the advice and predictive power of these elements. The IT environment can automatically take the right actions based on the available information and the knowledge of what is happening in the environment.
5. **Autonomic level**—the IT infrastructure operation is governed by business policies and objectives. Users interact with autonomic technology to monitor business processes, alter objectives or both.

*"The Tivoli group at IBM is introducing autonomic computing in a step-by-step, phased manner. This allows customers to leverage their existing technical and staff resources with today's autonomic offerings and enables them to progress to the next level as new offerings appear. This process can help customers achieve a steadily increasing return on their investments."*

—Paul Mason, IDC

**Autonomic computing best practices**

The evolution to autonomic computing is not just about improved technologies or tools; changes are needed in skill levels, processes and benchmarks to evolve and move toward autonomic computing. These are represented in the diagram below.

| Basic<br>Level 1 | Managed<br>Level 2 | Predictive<br>Level 3 | Adaptive<br>Level 4 | Autonomic<br>Level 5 |
|---|---|---|---|---|
| **Process**<br>Informal, reactive, manual | **Process**<br>Documented, improved over time, leverage of industry best practices, manual process to review IT performance | **Process**<br>Proactive, shorter approval cycle | **Process**<br>Automation of many resource management best practices and transaction management best practices, driven by service-level agreements | **Process**<br>IT service management and IT resource management best practices are automated |
| **Tools**<br>Local, platform and product-specific | **Tools**<br>Consolidated resource management consoles with correlation of events, problem management system, automated software install, intrusion detection, load balancing | **Tools**<br>Role-based consoles with analysis and recommendations; product configuration advisors; realtime view of current and future IT performance; automation of some repetitive tasks; common knowledge base of inventory and dependency management | **Tools**<br>Policy-management tools drive dynamic change based on resource-specific policies | **Tools**<br>Costing/financial analysis tools, business and IT modeling tools, tradeoff analysis; automation of some e-business management roles |
| **Skills**<br>Platform-specific, geographically dispersed with technology | **Skills**<br>Multiple platform skills, multiple management tool skills | **Skills**<br>Cross-platform business system knowledge, IT workload management skills, some business-process knowledge | **Skills**<br>Service objectives and delivery per resource, analysis of impact on business objectives | **Skills**<br>e-business cost and benefit analysis, performance modeling, advanced use of financial tools for IT context |
| **Benchmarks**<br>Time to fix problems and finish tasks | **Benchmarks**<br>System availability, time to close trouble tickets and work requests | **Benchmarks**<br>Business system availability, service-level agreement attainment, customer satisfaction | **Benchmarks**<br>Business system response time, service-level agreement attainment, customer satisfaction, IT contribution to business success | **Benchmarks**<br>Business success, competitiveness of service-level agreement metrics, business responsiveness |

*"By focusing on staged delivery and what is actually available today and tomorrow, Tivoli software can help its customers work toward the autonomic computing vision."*

—James Governor, Illuminata

As companies progress through the five levels of autonomic computing, the processes, tools and benchmarks become increasingly sophisticated, and the skills requirement becomes more closely aligned with the business.

At the basic level, if IT organizations are formally measured at all, they are typically evaluated on the time required to finish major tasks and fix major problems. The IT organization is viewed as a cost center, with variable labor costs preferred over an investment in centrally coordinated systems management tools and processes.

At the managed level IT organizations are measured on the availability of their managed resources, their time to close trouble tickets in their problem management system and their time to complete formally tracked work requests. To improve on these measurements, IT organizations document their processes and continually improve them through manual feedback loops and adoption of best practices. IT organizations gain efficiency through consolidation of management tools to a set of strategic platforms and through a hierarchical problem management triage organization.

In the predictive level IT organizations are measured on the availability and performance of their business systems and their return on investment. To improve, IT organizations measure, manage and analyze transaction performance. The critical nature of the IT organization's role in business success is understood. Predictive tools are used to project future IT performance, and many tools make recommendations to improve future performance.

In the adaptive level IT resources are automatically provisioned and tuned to optimize transaction performance. Business policies, business priorities and service-level agreements guide the autonomic infrastructure behavior. IT organizations are measured on comprehensive business system response times (transaction performance), the degree of efficiency of the IT infrastructure and their ability to adapt to shifting workloads.

In the autonomic level IT organizations are measured on their ability to make the business successful. To improve business measurements they understand the financial metrics associated with e-business activities and supporting IT activities. Advanced modeling techniques are used to optimize e-business performance and quickly deploy newly optimized e-business solutions.

What should you do next in assessing your self-managing environment? Now that we described what autonomic computing is and how you can benefit from it, your next step is to assess your own environment from a self-management perspective. IBM can help you do just that.

**Leveraging open standards**

Open standards are essential for managing resources and processes across the system hierarchy layers. IBM leverages its leadership in open standards as the foundation for its autonomic system management solution. These standards currently include:

- *Java™ Management Extensions*
- *Web service-level agreements*
- *Storage Networking Industry Association*
- *Open-grid systems architecture*
- *Web Services standards*
- *Telecom management*
- *Distributed Management Taskforce*
- *Common Information Model*
- *Internet Engineering Taskforce (Policy, Simple Network Management Protocol)*
- *Organization for the Advancement of Structured Information Standards (OASIS)*
- *Microsoft® Windows® management instrumentation*

**IBM products help make autonomic computing a reality today**

Tivoli® software from IBM provides a head start toward reaching the ultimate objective of a fully autonomic system and continues to participate in developing the ultimate solution. Many Tivoli management disciplines provide some level of automation that can help you achieve the potential benefits of implementing an autonomic self-managing environment. Tivoli products support and incorporate the autonomic behaviors of self-configuring, self-healing, self-optimizing and self-protecting. As of October 2002 Tivoli software has more than 20 products with autonomic capabilities. Following are just a few examples of these products.

***IBM Tivoli Configuration Manager*** is an integrated inventory and software distribution solution that offers multiple autonomic computing capabilities. Self-configuration is possible by using software package reference models to match desired system configurations. Inventory recognition of software inconsistencies, granular inventory scanning of specific system elements and built-in bandwidth throttling demonstrate the self-optimizing characteristics of Tivoli Configuration Manager. Built-in bandwidth throttling optimizes the rate at which software is distributed across networks, based on available bandwidth. With a checkpoint-restart feature, Tivoli Configuration Manager is self-healing in that software distributions which do not complete themselves are reinitiated at the point of failure rather than from the beginning, saving time and network usage.

***IBM Tivoli Storage Resource Manager*** can scan and discover storage resources in the IT environment. It supports policy-based automation for the allocation of storage quotas and storage space, monitors file systems and provides reports on capacity and storage asset utilization.

***IBM Tivoli Service Level Advisor*** proactively helps prevent SLA breaches with predictive capabilities. It performs trend analysis based on historical performance data from Tivoli Enterprise™ Data Warehouse and can predict when critical thresholds could be exceeded in the future. By sending an event to IBM Tivoli Enterprise Console®, self-optimizing actions can be taken to help prevent the problem from occurring.

***IBM Tivoli Identity Manager*** self-protects by centralizing identity management and integrating automated workflow with business processes while leveraging self-service interfaces to increase productivity. It provides role-based provisioning of users and access rights across a wide range of system and application environments, based on security policies. It uses workflow technologies to allow the right administrators to approve the creation of different classes of IDs. The provisioning system communicates directly with access-control systems to create accounts, supply user information and passwords and define the entitlements of the account.

***IBM Tivoli Monitoring*** exemplifies how self-optimization capabilities can trigger self-configuration activities. Using templates applied against a resource model engine, Tivoli Monitoring provides tailored automation. For example, if a server with one application instance sees increasing usage of a buffer pool—instead of just reporting this as an event to a systems management console—new IBM autonomic technology is designed to see the increased usage, check processor utilization, check for available system memory and automatically increase the buffer pool for the application. As demand changes and usage of the buffer pool decreases, it automatically lowers the pool ceiling to return system memory to the server.

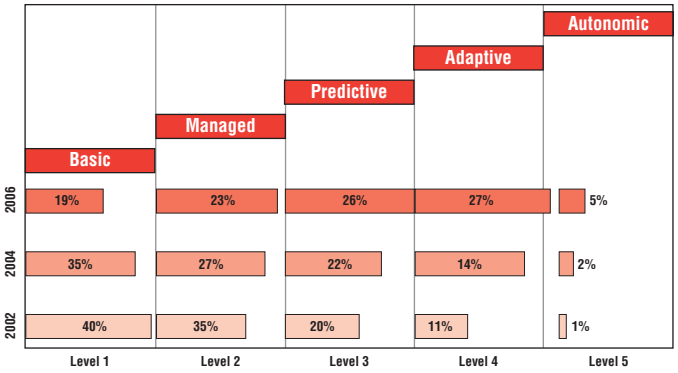
***IBM Tivoli Storage Manager*** includes several autonomic capabilities—self-configuration (automatic domain configuration and file-system identification), self-correcting (storage-pool restoration and correction), self-optimizing (backup and restore optimization) and self-protecting (automated protection of enterprise data across heterogeneous storage environments by initiating backup and archival processes). It provides policy-based automation capabilities, such as the ability to optimize disk and tape resources during a backup.

***IBM Tivoli Risk Manager*** offers a self-protecting core technology that is designed to shield IT infrastructure from cyberattacks by integrating many different solutions. Integration with antivirus, firewalls, virtual private networks, routers, Web servers and operating systems can help speed response to threats. For example, a typical large company might receive more than 300,000 heterogeneous security events per day from various security point products that need to be analyzed. Tivoli Risk Manager can help integrate and correlate the security information from the company's myriad security products to distill the same 300,000 events per day into 30 meaningful security alerts. This can help security administration teams focus on real security threats.

***IBM Tivoli Access Manager*** is a policy-based, access-control solution that supports comprehensive infrastructure protection. By extracting application security structures and enforcement logic into an encapsulated comprehensive security engine, Tivoli Access Manager makes user and vendor-developed applications self-protecting according to policy. Each transaction is examined for contextual characteristics in the application of policy and access rules.

**Are customers doing this today?**

Let's take a look at the mapping of the five autonomic computing levels and evaluate where customers are today and how future opportunity shows exciting growth potential for this evolutionary technology. In 2002, IBM internal estimates indicate that in the overall market 40 percent of customers are at the basic level, 35 percent at the managed level and 20 percent at the predictive level—illustrating the need for autonomic computing and the tremendous growth potential.




Beginning in 2003, the autonomic computing opportunity will be growing in the managed and predictive levels by 15 percent as customers begin to move out of the basic level and up the autonomic computing value ladder. Looking out to 2006 shows the highest growth in the predictive and analysis levels—both up 50 percent compared to 2002. Bottom line, autonomic computing is here today, and companies will continue to expand their breadth over the next few years.

*"Tripwire shares in the IBM vision of a future in which IT is more reliable and the lives of IT professionals more livable. We believe autonomic computing provides the means to make that vision a reality, and we are proud to partner with IBM in this endeavor."*

—W. Wyatt Starnes, founder and CEO of Tripwire, Inc.

**Summary**

Companies want and need to reduce their IT costs, simplify management of their IT resources, realize a fast return on their IT investment and provide high levels of availability, performance, security and asset utilization. Autonomic computing addresses these issues, and it is not just about new technology. This fundamental evolutionary shift in the way IT systems are managed will free the IT staff from detailed mundane tasks and allow them to focus on managing business processes. It can be accomplished through a combination of process changes, skills evolution, new technologies, architecture and open industry standards.

With autonomic computing reducing the demand for the specialized skills currently required to manage IT initiatives, projects are more likely to be implemented on time and on budget. The incorporation of autonomic technologies will lead to systems that deliver the expected quality of service, helping eliminate many configuration problems introduced in today's environments due to human error, reducing the man outages caused by malfunctions that can now be self-corrected and delivering maximum results and performance by continually optimizing the use of resources.

**To learn more**

For information on the IBM autonomic computing strategy and integrated solutions from IBM, contact your IBM sales representative or visit **ibm.com/tivoli**

## *Why Tivoli software products for Autonomic Computing?*

- *Because Tivoli software delivers autonomic capability today*
- *Because Tivoli software frees customers to focus on their business instead of their infrastructure*
- *Because Tivoli software combines the agility of a startup with the experience, quality and resources of IBM*
- *Because Tivoli software provides step-by-step roadmaps*
- *Because Tivoli software represents evolution not revolution*
- *Because the Tivoli software team is ready to assist (IBM and IBM Business Partners)*



IBM Corporation
Software Group
Route 100
Somers, NY 10589
U.S.A.

Printed in the United States of America
10-02


The Tivoli home page on the Internet can be found at **ibm.com/tivoli**

The IBM home page on the Internet can be found at **ibm.com**

♻ Printed in the United States on recycled paper containing 10% recovered post-consumer fiber.

G325-6869-00