A

Seminar report

On

Autonomic Computing

Submitted in partial fulfillment of the requirement for the award of degree
Of CSE

SUBMITTED TO: **SUBMITTED BY:**

www.studymafia.org www.studymafia.org

Acknowledgement

I would like to thank respected Mr…….. and Mr. ……..for giving me such a wonderful opportunity to expand my knowledge for my own branch and giving me guidelines to present a seminar report. It helped me a lot to realize of what we study for.

Secondly, I would like to thank my parents who patiently helped me as i went through my work and helped to modify and eliminate some of the irrelevant or un-necessary stuffs.

Thirdly, I would like to thank my friends who helped me to make my work more organized and well-stacked till the end.

Next, I would thank Microsoft for developing such a wonderful tool like MS Word. It helped my work a lot to remain error-free.

Last but clearly not the least, I would thank The Almighty for giving me strength to complete my report on time.

Preface

I have made this report file on the topic **Autonomic Computing**; I have tried my best to elucidate all the relevant detail to the topic to be included in the report. While in the beginning I have tried to give a general view about this topic.

My efforts and wholehearted co-corporation of each and everyone has ended on a successful note. I express my sincere gratitude to …………..who assisting me throughout the preparation of this topic. I thank him for providing me the reinforcement, confidence and most importantly the track for the topic whenever I needed it.

CONTENTS

INTRODUCTION

"Civilization advances by extending the number of important operations which we can perform without thinking about them." - Alfred North Whitehead

This quote made by the preeminent mathematician Alfred Whitehead holds both the lock and the key to the next era of computing. It implies a threshold moment surpassed only after humans have been able to automate increasingly complex tasks in order to achieve forward momentum.

There is every reason to believe that we are at just such a threshold right now in computing. 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 shortage of skilled I/T workers to manage all of the systems. The high-tech industry has spent decades creating computer systems with ever- mounting degrees of complexity to solve a wide variety of business problems. Ironically, complexity itself has become part of the problem. It's a problem that's not going away, but will grow exponentially, just as our dependence on technology has.

But as Mr. Whitehead so eloquently put it nearly a century ago, 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 nervous systems 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.

WHAT IS AUTONOMIC COMPUTING?

"Autonomic Computing" is a new vision of computing initiated by IBM. This new paradigm shifts the fundamental definition of the technology age from one of computing, to one defined by data. Access to data from multiple, distributed sources, in addition to traditional centralized storage devices will allow users to transparently access information when and where they need it. At the same time, this new view of computing will necessitate changing the industry's focus on processing speed and storage to one of developing distributed networks that are largely self-managing, self-diagnostic, and transparent to the user.

The term autonomic is derived from human biology. The autonomic nervous system monitors our heartbeat, checks our blood sugar level and keeps our body temperature close to 98.6 °F, without any conscious effort on our part. In much the same way, autonomic computing components anticipate computer system needs and resolve problems —with minimal human intervention. However, there is an important distinction between autonomic activity in the human body and autonomic responses in computer systems. Many of the decisions made by autonomic elements in the body are involuntary, whereas autonomic elements in computer systems make decisions based on tasks you choose to delegate to the technology. In other words, adaptable policy — rather than rigid hard coding determines the types of decisions and actions autonomic elements make in computer systems.

Key Elements of Autonomic Computing

The elements of autonomic computing can be summarized in to 8 key points.

Knows Itself

An autonomic computing system needs to "know itself" - its components must 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 to 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.

Configure Itself

An autonomic computing system must configure and reconfigure itself under varying (and in the future, even unpredictable) conditions. System configuration or "setup" must occur automatically, as well as dynamic adjustments to that configuration to best handle changing environments

Optimies Itself

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.

Heal Itself

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.

Protect Itself

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

Adapt Itself

An autonomic computing system must know its environment and the context surrounding its activity, and act accordingly. It 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.

Open Itself

An autonomic computing system cannot exist in a hermetic environment. While independent in its ability to manage itself, it must function in a heterogeneous world and implement open standards -- in other words, an autonomic computing system cannot, by definition, be a proprietary solution.

Hide Itself

An autonomic computing system will anticipate the optimized resources needed while keeping its complexity hidden. It must marshal I/T resources to shrink the gap between the business or personal goals of the user, and the I/T implementation necessary to achieve those goals -- without involving the user in that implementation

Autonomic Computing and Current Computing-A Comparison

In an autonomic environment, system components —from hardware such as desktop computers and mainframes to software such as operating systems and business applications — are self-configuring, self-healing, self-optimizing and self- protecting. These self-managing attributes can be compared as given in the table.

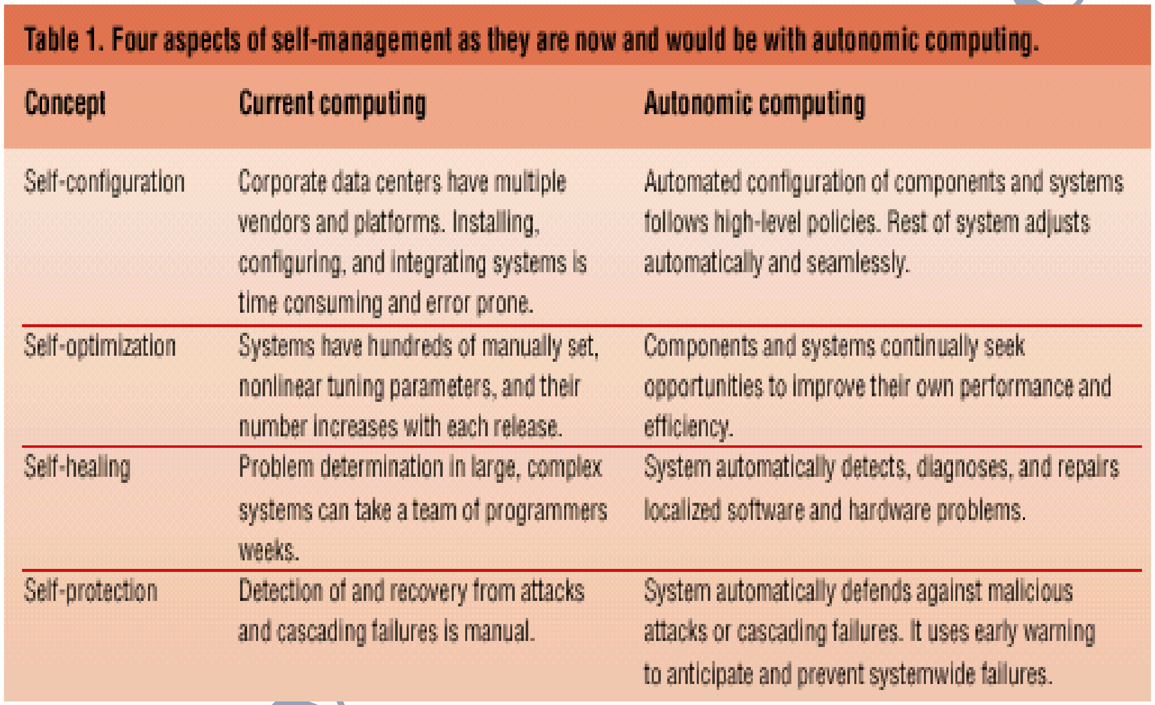


Table 1. Four aspects of self-management as they are now and would be with autonomic computing.

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.

AUTONOMIC COMPUTING ARCHITECTURE

The autonomic computing architecture concepts provide a mechanism discussing, comparing and contrasting the approaches different vendors use to deliver self-managing attributes in an autonomic computing system.The autonomic computing architecture starts from the premise that implementing self-managing attributes involves an intelligent control loop. This loop collects information from the system. makes decisions and then adjusts the system as necessary. An intelligent control loop can enable the system to do such things as:

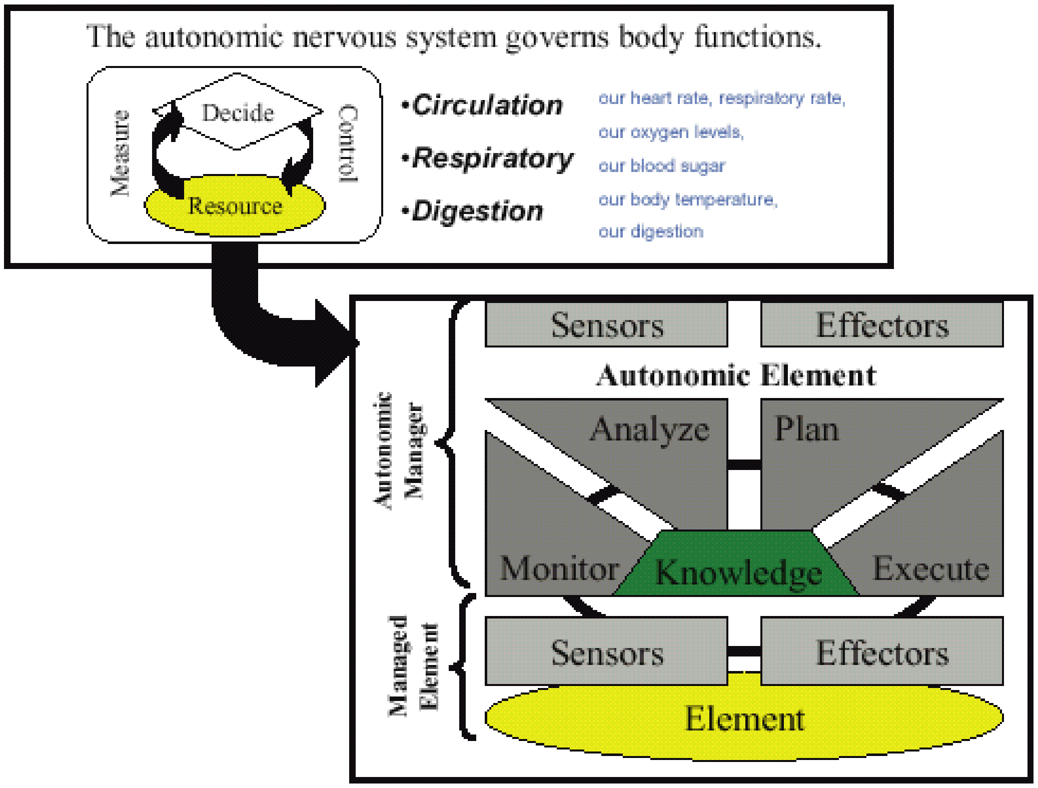
- Self-configure, by installing software when it detects that software is missing
- Self-heal, by restarting a failed element
- Self-optimize, by adjusting the current workload when it observes an increase in capacity
- Self-protect, by taking resources offline if it detects an intrusion attempt.

Control loops

A control loop can be provided by a resource provider, which embeds a loop in the runtime environment for a particular resource. In this case, the control loop is configured through the manageability interface provided for that resource (for example, a hard drive).In some cases, the control loop may be hard-wired or hard coded so it is not visible through the manageability interface.

Autonomic systems will be interactive collections of *autonomic elements*—individual system constituents that contain resources and deliver services to humans and other autonomic elements. , An autonomic element will typically consist of one or more managed elements coupled with a single autonomic manager that controls and represents them.

In an autonomic environment, autonomic elements work together, communicating with each other and with high-level management tools. They regulate themselves and, sometimes, each other. They can proactively manage the system, while hiding the inherent complexity of these activities from end users and IT professionals. Another aspect of the autonomic computing architecture is shown in the diagram below. This portion of the architecture details the functions that can be provided for the control loops. The architecture organizes the control loops into two major elements —a managed element and an autonomic manager. A managed element is what the autonomic manager is controlling. An autonomic manager is a component that implements a control loop.



In an autonomic computing architecture, control loops facilitate system management

Managed Elements

The managed element is a controlled system component. 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, 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 managed element is controlled through its sensors and effectors:

- The sensors provide mechanisms to collect information about the state and state transition of an element. To implement the sensors, you can either use a set of “get ”operations to retrieve information about the current state, or a set of management events (unsolicited, asynchronous messages or notifications)that flow when the state of the element changes in a significant way.
- The effectors are mechanisms that change the state (configuration) of an element. In other words, the effectors are a collection of “set ”commands or application programming interfaces (APIs)that change the configuration of the managed resource in some important way.

The combination of sensors and effectors form the manageability interface that is available to an autonomic manager. As shown in the figure above, by the black lines connecting the elements on the sensors and effectors sides of the diagram, the architecture encourages the idea that sensors and effectors are linked together. For example, a configuration change that occurs through effectors should be reflected as a configuration change notification through the sensor interface.

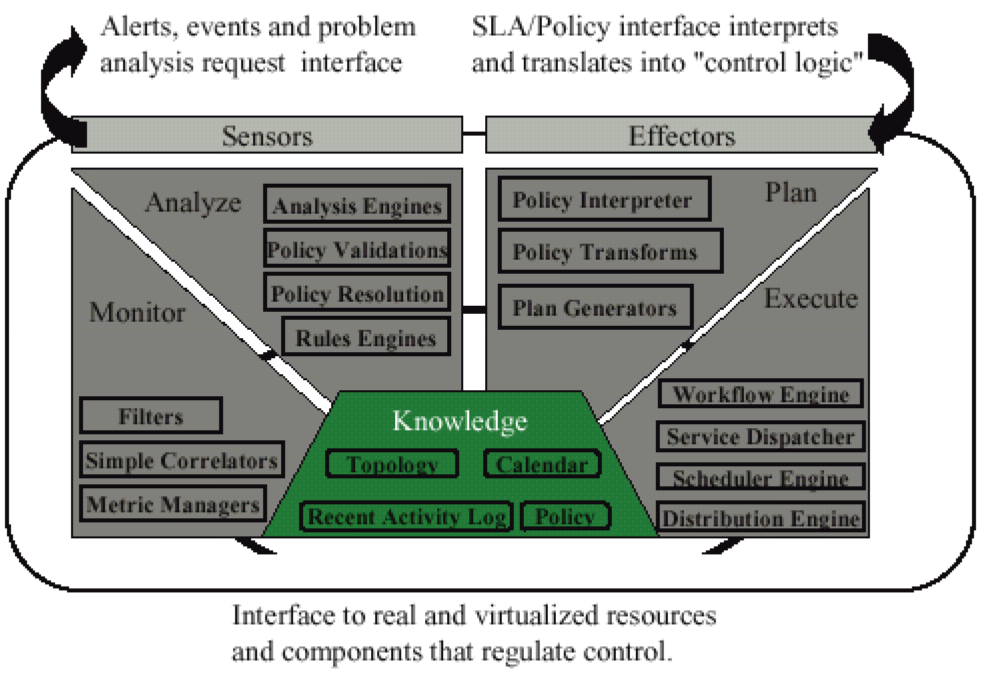
Autonomic manager

The autonomic manager is a component that implements the control loop. 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.

The architecture dissects the loop into four parts that share knowledge:

- The monitor part provides the mechanisms that collect, aggregate, filter, manage and report details (metrics and topologies) collected from an element.
- The analyze part provides the mechanisms to correlate and model complex situations (time-series forecasting and queuing models, for example). These mechanisms allow the autonomic manager to learn about the IT environment and help predict future situations.
- The plan part provides the mechanisms to structure the action needed to achieve goals and objectives. The planning mechanism uses policy information to guide its work.
- The execute part provides the mechanisms that control the execution of a plan with considerations for on-the-fly updates.

The following diagram provides a more detailed view of these four parts by highlighting some of the functions each part uses.

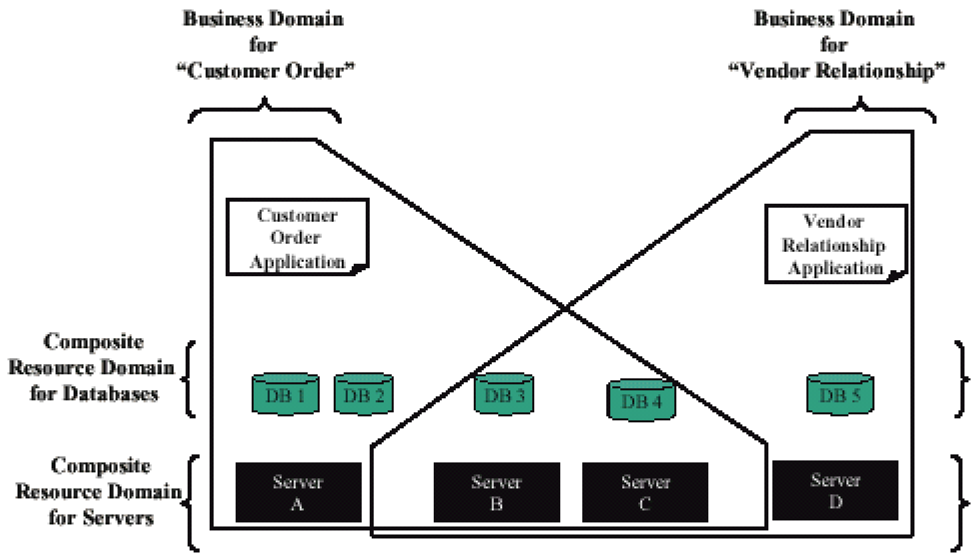


The functional details of an autonomic manager

The four parts work together to provide the control loop functionality. The diagram shows a structural arrangement of the parts —not a control flow. The bold line that connects the four parts should be thought of as a common messaging bus rather than a strict control flow. In other words, there can be situations where the plan part may ask the monitor part to collect more or less information. There could also be situations where the monitor part may trigger the plan part to create a new plan. The four parts collaborate using asynchronous communication techniques, like a messaging bus.

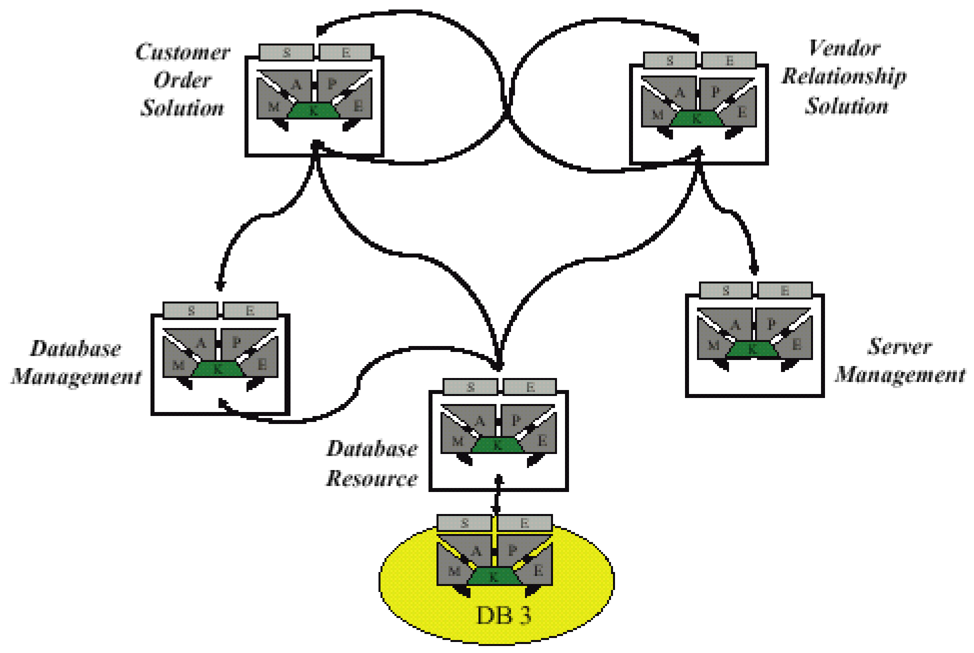
Autonomic manager collaboration

The following diagram shows an example of a simple IT system that includes two business applications: a customer order application and a vendor relationship application. Separate teams manage these applications. Each of these applications depends on a set of IT resources —databases and servers —to deliver its functionality. Some of which are shared resources —DB 3,DB 4,Server B and Server C —are shared between the applications, we managed separately. There is a minimum of four management domains (decision-making contexts)in this example. Each of the applications (customer order and vendor relationship)has a domain, focused on the business system it implements. In addition, there is a composite resource domain for managing the common issues across the databases and a composite resource domain for managing common issues for the servers.


Business Domain for "Customer Order"
Business Domain for "Vendor Relationship"
Customer Order Application
Vendor Relationship Application
Composite Resource Domain for Databases
Composite Resource Domain for Servers
DB 1
DB 2
DB 3
DB 4
DB 5
Server A
Server B
Server C
Server D

IT systems can share resources to increase efficiency

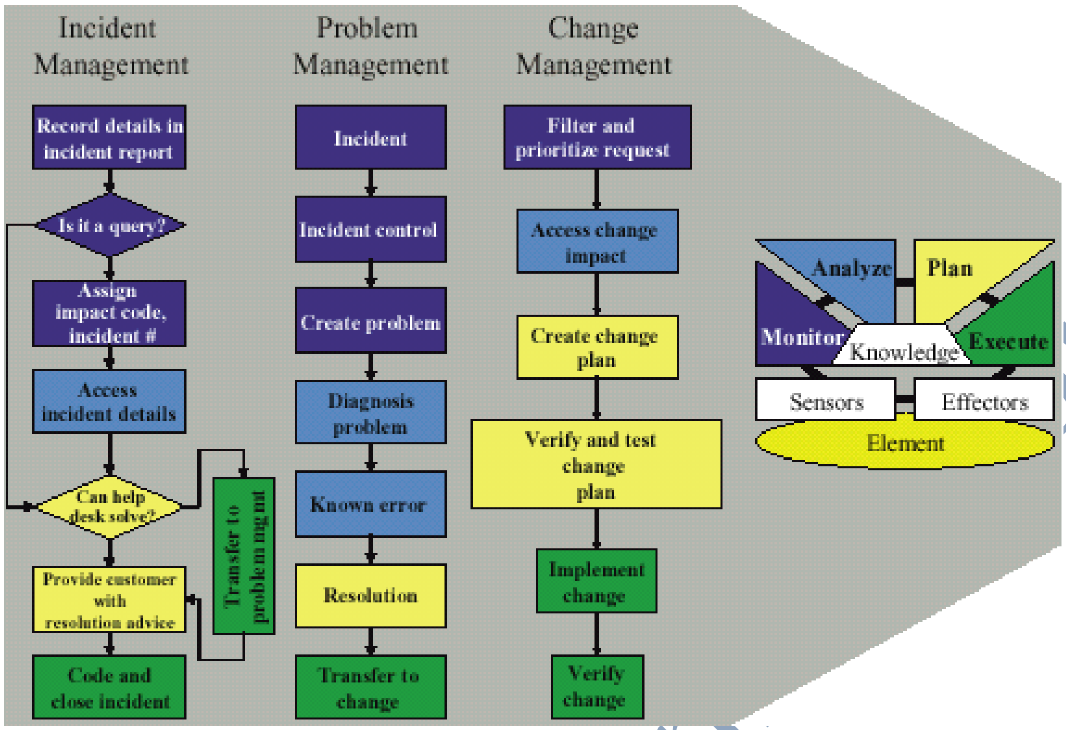
Now, let us apply the autonomic computing architecture to this example, to see how the autonomic managers would be used. The following diagram illustrates some of the autonomic managers that either directly or indirectly manage DB 3 and some of the interaction between these autonomic managers. There are six autonomic managers in this illustration: one for each of the management domains, one embedded in the DB 3 resource and one dedicated to the specific database resource. Since the decision-making contexts for these autonomic managers are interdependent and self-optimizing, the autonomic managers for the various contexts will need to cooperate. This is accomplished through the sensors and effectors for the autonomic managers, using a "matrix management protocol." This protocol makes it possible to identify situations in which there are "multiple managers "situations and enables autonomic managers to electronically negotiate resolutions for domain conflicts, based on a system wide business and resource optimization policy.



Six autonomic managers directly and indirectly manage the DB3 resource

Self-managing systems change the IT business

The mechanics and details of IT processes, such as change management and problem management, are different, but it is possible to categorize 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 parts of the architecture. The approximate relationship between the activities in some IT processes and the parts of the autonomic manager are illustrated in the following figure.



How autonomic computing affects IT processes

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 required of the IT professional. 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.

A GRAND CHALLENGE

A Grand Challenge is 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.The difficulty in developing and implementing autonomic computing is daunting enough to constitute a Grand Challenge. At the heart of the matter is the need to bring together minds from multiple technical and scientific disciplines as well as differentiated businesses and institutions to share a sense of urgency and purpose.

Part of the challenge lies in the fact that autonomic computing has been conceived as a holistic approach to computing. The difficulty is not the machines themselves. Year after year scientists and engineers have brilliantly exceeded goals for computer performance and speed. The problem now lies in creating the open standards and new technologies needed for systems to interact effectively, to enact pre-determined business policies more effectively, and to be able to protect themselves and "heal" themselves with a minimal dependence on traditional I/T support. This broader systems view has many implications:

On a conceptual level, the way we define and design computing systems will need to change:

- The computing paradigm will change from one based on computational power to one driven by data.
- The way we measure computing performance will change from processor speed to the immediacy of the response.
- Individual computers will become less important than more granular and dispersed computing attributes.
- The economics of computing will evolve to better reflect actual usage - what IBM calls e-sourcing.

Based on new autonomic computing parameters the functionality of individual components will change and may include:

- Scalable storage and processing power to accommodate the shifting needs of individual and multiple autonomic systems.

- Transparency in routing and formatting data to variable devices
- Evolving chip development to better leverage memory
- Improving network-monitoring functions to protect security, detect potential threats and achieve a level of decision-making that allows for the redirection of key activities or data.
- Smarter microprocessors that can detect errors and anticipate failures These are just some of the implications and resulting challenges that lie ahead.

ENGINEERING CHALLENGES

Virtually every following aspect of autonomic computing offers significant engineering challenges.

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. 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 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 Integration6 or an Open Grid Services Architecture (OGSA) registry, providing a description of its capabilities and details about 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. 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 effort7 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. 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 Computer 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 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.
- 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 colleagues9 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 colleagues10 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 defi- nitions 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 singleagent 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.

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.

RELATED TECHNOLOGIES

ROLE OF CURRENT TECHNOLOGIES

The following table gives a summary of some current components and their proposed development under autonomic computing.

Levels of sophistication	Known examples	Current directions	Future goal
Serving the world (i.e., people, business processes)	SMS	Policy management, Storage tank	Policy language and protocols
Heterogeneous components interacting	SNMP	Mounties, Workload management	Autonomic computing stack, Social policy, DB/storage co-optimization
Homogeneous components interacting	Adaptive network routing, network congestion control, high availability clustering	Collective intelligence, Storage Bricks, Oceano	New packaging concepts for storage, Subscription computing
Components	ESS, RAID, DB optimizer, virus management	eLiza, SMART/LEO, Software rejuvenation	More of the same and better

For “Components” one example is LEO (Learning in Query Optimization), a DB optimizer that learns from past performance. It will be in the next version of DB2.

In the “Homogeneous components interacting” row, the following examples can be included:

1. Intelligent storage bricks: The idea is to have higher redundancy than RAID, protection of performance hot spots with proactive copies (by the system), and elimination of repair for life of system by building extra drives into the system.
2. New packaging concepts for storage: The idea is to change the packaging of an array of disks from a 2-D grid to a 3-D cube. Why haven’t we done this before? Because you need to get at each drive so that you can repair or replace it. Again, what you do here is to build extra capacity into the system so that when one drive fails, it won’t be missed. Another problem is heat but they’ve been able to address this. There is a prototype of this called the IceCube which is basically the size of a medium-sized packing box. It can store up to 1 Petabyte (10^15bytes), 250kW, 75dB air noise. Should last for 5 years without any service ever.

In the ‘heterogeneous components interacting” there is Mounties system which enables goal-oriented recovery from system failure instead of procedural oriented recovery.

In the “serving the world” policy managed storage in a system called the Storage TankFor every file or folder, the user sets policies of availability, security, and performance. The system figures out where to put the data, what level of redundancy, what level of backup, etc. This is goal-oriented management.

TECNOLOGIES DEVELOPED BY IBM

IBM has developed four technologies related to autonomic computing

Log and Trace Tool

Log and Trace tool is used for problem determination, which helps to take autonomic systems from figuring out the problems to debugging applications and middleware.The Log and Trace Analyzer 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 Analyzer for Autonomic Computing works with ISV plug-ins.

It works as follows.

It provides a consolidated environment that deals with logs and traces produced by various components of a deployed system. This technology links these two sets of tools (tracing and logging) and helps bridge the gap between determination of problems and debugging of applications and middleware. By capturing and correlating events from end-to-end execution in the distributed stack, this tool allows for a more structured analysis of distributed application problems that facilitates the development of autonomic, self-healing, and self-optimizing capabilities.It is supported in the folllowing platforms:- Windows NT; Linux; Windows 2000; Win XP

Agent Building and Learning Environment (ABLE)

ABLE is a Java framework, component library, and productivity tool kit for building intelligent agents using machine learning and reasoning. The ABLE research project is made available by the IBM T. J. Watson Research Center. ABLE (Agent Building and Learning Environment) Rules Engine is used for Complex Analysis, which uses a set of algorithms that allows intelligent agents to capture data, and can predict future steps to take based on system experienceThe ABLE framework provides a set of Java interfaces and base classes used to build a library of JavaBeans called AbleBeans. The library includes AbleBeans for reading and writing text and database data, for data transformation and scaling, for rule-based inferencing using Boolean and fuzzy logic, and for machine learning techniques such as neural networks, Bayesian classifiers, and decision trees. Developers can extend the provided AbleBeans or implement their own custom algorithms. Rule sets created using the ABLE Rule Language can be used by any of the provided inferencing engines, which range from simple if-then scripting to light-weight inferencing to heavy-weight AI algorithms using pattern matching and unification. Java objects can be created and manipulated using ABLE rules. User defined functions can be invoked from rules to enable external data to be read and actions to be invoked.

It works as follows.

Core beans may be combined to create function-specific JavaBeans called AbleAgents. Developers can implement their own AbleBeans and AbleAgents and plug them into ABLE's Agent Editor. Graphical and text inspectors are provided in the Agent Editor so that bean input, properties, and output can be viewed as machine learning progresses or as values change in response to methods invoked in the interactive development environment.

Application-level agents can be constructed from AbleBean and AbleAgent components using the ABLE Agent Editor or a commercial bean builder environment. AbleBeans can be called directly from applications or can run autonomously on their own thread. Events can be used to pass data or invoke methods and can be processed in a synchronous or asynchronous manner. The distributed AbleBeans and AbleAgents are as follows:

Data beans

- AbleImport reads data from flat text files.
- AbleDBImport reads data from SQL databases.
- AbleFilter filters, transforms, and scales data using translate template specifications.
- AbleExport writes data to flat text files.
- AbleTimeSeriesFilter collects periods of data for use in predicting future values.

Learning beans

- Back Propagation implements enhanced back propagation algorithm used for classification and prediction.
- Decision tree creates a decision tree for classification.
- Naive Bayes learns a probabalistic model for classification.
- Radial Basis Function uses radial basis functions to adjust weights in a single, hidden-layer neural network for prediction.
- Self-Organizing Map clusters data using Gaussian neighborhood function.
- Temporal Difference Learning uses reinforcement learning for time series forecasting; gradient descent is used to adjust network weights.

Rules beans inferencing engines include

- Backward chaining
- Forward chaining
- Forward chaining with working memory
- Forward chaining with working memory and Rete'-based pattern matching
  Predicate logic
- Fuzzy logic
- Script

Agents

- Genetic search manipulates a population of genetic objects which may include

AbleBeans.

- Neural classifier uses back propagation to classify data.
- Neural clustering uses self-organizing maps to segment data.
- Neural prediction uses back propagation to build regression models.
- Script uses rule sets to define its init, process, and timer actions.
- JavaScript names JavaScripts to run when the agent's init, process, or time actions are called.

Platforms: Windows NT; Windows 95; Windows 98; Linux; UNIX; OS/2; 2000; OS/400

For an application developer, ABLE makes your life easier by providing a set of intelligent beans and an editor for combining them into agents. The ABLE design philosophy is to provide a set of reusable Java components (JavaBeans) and a light-weight framework for combining them to build intelligent components (agents). ABLE provides Sensors and Effectors to allow application developers to easily plug the agent into their Java application environment.

If you are doing research on intelligent agents, ABLE makes your life easier by providing a flexible Java framework for combining the ABLE beans with your algorithms or ideas about how agents should be constructed. The ABLE Editor provides a nice GUI environment for building and testing custom learning and reasoning algorithms. The ABLE Inspectors use standard Java introspection to display state information about the custom beans you create, and they provide a way to view the data in graphical form such as bar charts, line plots, and time plots. It is fairly easy to take an existing Java algorithm and package it as an AbleBean for use with the rest of the AbleBeans provided in the ABLE tool kit.

Monitoring Engine

Monitoring Engine, now available in IBM Tivoli Monitoring, enables root-cause analysis for IT failures, server-level correlation of multiple IT systems, and automated corrective measures.

With the complimentary IBM Tivoli Monitoring Resource Model Builder, it has become easier to create Automated Best Practices for monitoring and delivering autonomic self healing capabilities for IT systems. Resource Models are the building blocks for monitoring and healing your IT environment via Automated Best Practices. They contain specific metrics, events, thresholds and parameters which are used to determine the health of IT resources along with specifications for corrective actions in the event of failures.

The downloadable IBM Tivoli Monitoring Resource Model Builder provides a standard Eclipse interface providing a simple step by step wizard to build resource models. A development organization can leverage its core competencies to easily specify Automated Best Practices for monitoring and automatic curing for IT systems right out of the box. Working with the IBM Tivoli Monitoring family of products, the IBM Tivoli Resource Model Builder allows quickly deliver greater value to to customers with better managed IT systems.

System requirements: Windows XP or Windows 2000 .

Business Workload Manager

Business Workload Management for Heterogeneous Environments, which will be soon out in IBM Tivoli Monitoring for Transaction Performance Version 5.2 , uses the ARM (Application Resource Measurement) standard to determine why bottlenecks happen, using response time measurement, transaction processing segment reporting, and learning of transaction workflow through servers and middleware. The software then adjusts resources to avoid bottlenecks.

Business Workload Manager (BWLM) Prototype is a technology that enables instrumentation of applications with Application Response Measurement (ARM) in order to monitor the performance of transactions across a distributed environment. This ARM-based performance information will be used by BWLM to monitor and adjust the allocation of computing resources on an ongoing, split-second basis. Planned functions include the ability of BWLM to detect changes in its environment and decide which resources (system, network, load-balancing patterns) to adjust in order to enable a network of systems to meet end-to-end performance goals. When middleware (or, in this prototype, an application) is instrumented with ARM, it will be able to take advantage of products such as BWLM and participate in IBM's autonomic computing initiative.

The prototype allows one to observe and build upon the instrumented application using the ARM 4.0 (pre-approval version) standard to handle workload management for better transaction flow across systems and applications. This technology will provide significant value in understanding response times and transaction flow for the following:

- improved service-level management based on performance policies
- determining where transactions hang
- active workload management for better capacity use
- understanding bottlenecks for better capacity planning.

The prototype demonstrates instrumentation of an application (in this case, a PlantsByWebsphere EJB) using ARM APIs which could otherwise have been achieved by instrumenting middleware such as WebSphere. The use of service classes to define performance policies is also shown.

In addition to the prototype, there is a BWLM demonstration. The BWLM Demo includes a simulation of BWLM's administrative user interface, which is used to show how performance policies are set, to locate performance bottlenecks, and to see how the software optimizes workloads by reallocating resources.

The goals of the demo are to communicate workload management concepts and to show some of the functionality planned for the product. Indeed, the BWLM Demo shows capabilities and functions that are not part of the prototype. The prototype is an example of instrumentation of an application so that it exploits a subset of BWLM capabilities.
It works as folllows

The pre-approval ARM 4.0 standard supports both C and Java applications and will allow developers to instrument applications so that they collect performance data. This technology is intended to drive the first significant ARM instrumentation in commercial middleware. This basic prototype will feature the following:

- Administrative application
- Management server and BWLM agent for collecting ARM data

- Simple reporting
    - Server class reporting
    - Service class drill-down reporting
    - High-level server statistics

In the BWLM Demo, the user interface in no way reflects the look-and-feel of the user interface that will be provided with the actual product; the goals of the demo are to communicate workload management concepts and to show some of the functionality planned for the initial release. Additionally, the demo may not reflect the actual capabilities of any toolkit that may eventually be released by IBM. It works in Windows Platform.

OTHER RELATED TECHNOLOGIES

There are many other related technologies which are developed by various companies which will play active role in autonomic computing**.**

OGSA (Open Grid Systems Architecture) standard

OptimalGrid -- a research prototype middleware from -- aims to simplify creating and managing large-scale, connected, parallel grid applications. It optimizes performance and includes autonomic grid functionality. It's not a toolkit, and 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. If you're a Java programmer and you're working with a grid, or if your'e thinking about getting started with grid, read the first part of the tutorial. It describes OptimalGrid and the kinds of problems it's designed to handle. If you determine that, in fact, OptimalGrid and your problem are a good match, then you can complete the tutorial to download and install OptimalGrid and play with it.

Generic Log Adapter

Generic Log Adapter for Autonomic Computing is a rule-based tool that transforms software log events into the standard situational event formats in the autonomic computing architecture. The adapter is an approach to providing a producer proxy for the early participation of software groups in the autonomic computing architecture.

The Adapter consists of two main components: a Rule Builder and configuration tool; and the adapter run-time environment. In addition, the Adapter provides a plug-in architecture for customization with required functionality external to the user's software.
It works as follows.

The Rule Builder is used to generate parsing rules for a product's log file(s) and configuration of the Adapter. The rules and the product log(s) are fed into the adapter run-time environment, which converts the logs into the standard situation formats of the autonomic computing architecture, using appropriate schema, and forwards them to Log and Trace Analyzer and/or to any management tools capable of consuming the adapter's output.
It is supported in the following platforms.

Windows NT; Windows 2000; Windows XP

CONCLUSION

Is it possible to meet the grand challenge of autonomic computing without magic and without fully solving the AI problem? It is possible , 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.

No company is going to be able to control all the parts of an autonomic system. It's going to have to be an open source system because there are so m any parts. These parts will provide plenty of opportunity for competition.

REFERENCES

- www.google.com
- www.wikipedia.com
- www.studymafia.org