An Architectural Approach to Autonomic Computing

Steve R. White, James E. Hanson, Ian Whalley, David M. Chess, and Jeffrey O. Kephart
IBM Thomas J. Watson Research Center
{srwhite,jehanson,inw,chess,kephart}@us.ibm.com

Abstract

We describe an architectural approach to achieving the goals of autonomic computing. The architecture that we outline describes interfaces and behavioral requirements for individual system components, describes how interactions among components are established, and recommends design patterns that engender the desired system-level properties of self-configuration, self-optimization, self-healing and self-protection. We have validated many of these ideas in two prototype autonomic computing systems.

1. Introduction

Creating large-scale computing systems that manage themselves in accordance with high-level guidance from humans has been recognized as a grand challenge—one upon which the future of information technology rides [1, 2, 3].

Ultimately, our success in meeting this challenge will depend not only on our ability to invent new technologies, but also upon creating an architecture for self-management that exploits these technologies appropriately. Furthermore, we believe that the right architecture can, by itself, provide the key to achieving autonomic behavior at the system level. This paper attempts to motivate an architectural approach to autonomic computing.

An architecture for autonomic computing must accomplish two fundamental goals. First, it must describe the external interfaces and behaviors required to make an individual component autonomic, that is, self-managing. (We do not impose any requirements on the internal structure of these components.) Second, it must describe how to compose systems out of these autonomic components in such a way that the system as a whole is self-managing. We seek to do both.

We base our approach on a service-oriented architecture [4, 5]. Our approach bears much in common with agent-oriented systems [6] in that the system is composed of interacting, goal-driven autonomous or semi-autonomous components that sense and respond to their environment. It goes beyond both of these frameworks in that we specify the interfaces, behaviors and design patterns that are required to achieve self-management.

A key concept in our architecture is the **autonomic element**. Following [2], we define an autonomic element as a component that is responsible for managing its own behavior in accordance with policies, and for interacting with other autonomic elements to provide or consume computational services.

In our approach, every component of an autonomic system is an autonomic element. This includes computing resources such as a database, a storage system, or a server. It includes higher-level elements with some management authority, such as a workload manager or a provisioner. It also includes elements that assist others in doing their tasks, such as a policy repository, a sentinel, a broker, or a registry.

This paper represents a first step towards describing an architecture for autonomic computing. In Section 2, we discuss required behavioral properties of autonomic elements. In Section 3, we describe interfaces and interactions among autonomic elements. In Section 4, we describe how to build a system with autonomic behaviors, starting with a collection of autonomic elements and, in Section 5, discuss common design patterns for doing so. Finally, in Section 6, we summarize prototype autonomic computing systems that have proven helpful in developing, verifying, and refining the architecture.

2. Autonomic element behaviors

In this section, we discuss some of the required and optional behaviors that we demand of autonomic elements. Because of its importance, the use of policies is discussed separately in Section 2.3.

Proceedings of the International Conference on Autonomic Computing (ICAC'04)
0-7695-2114-2/04 $20.00 © 2004 **IEEE**

2.1 Required behaviors

First, an autonomic element *must* be self-managing—that is, it must be responsible for configuring itself internally, for healing over internal failures, for optimizing its own behavior, and for protecting itself from external probing and attack. In order to simplify systems management in the large, an autonomic element *must* handle problems locally, whenever possible. For example, if it discovers that an autonomic element upon which it relies for service is not abiding by its agreement, it must try to resolve the problem. It may do so by demanding that the other element provide the agreed-upon service, or by terminating the relationship and finding another, more suitable element to provide the service.

Second, an autonomic element *must* be capable of establishing and maintaining relationships with other autonomic elements—those to which it provides service, and those which provide service to it. This in turn induces several requirements. An autonomic element *must* describe its service accurately and in such a way that it is accessible and understandable to other autonomic elements. Relationships, as we will see, are based upon agreements, so the autonomic element *must* understand and abide by the terms of its agreements. Additionally, the autonomic element *must* be capable of negotiating (even trivially) to establish agreements.

Third, an autonomic element *must* manage its behavior and relationships so as to meet its obligations, either by appropriately tuning or configuring its own parameters, or by drawing upon the resources of other autonomic elements in the system. There are two types of obligations to which an autonomic element may be subject. First, an autonomic element *must* honor the terms of its agreements. Second, an autonomic element *must* be capable of receiving and abiding by policies (cf. Section 2.3). An autonomic element *must* reject any service request that would violate its policies or agreements. Similarly, it *must* refuse (or counterpropose) any proposed relationship or policy that would cause a violation of its existing relationships or policies. It *must* have sufficient analytic capabilities to support these functions.

Administrative relationships are not treated specially. An autonomic element receives a directive from another element in the same manner that it receives ordinary requests. If, upon checking its policy, it discovers that the requestor has sufficient authority to warrant taking the request as a command, then it acts accordingly. If the autonomic element receives conflicting requests from two autonomic elements that manage it, the autonomic element itself has the responsibility for resolving the conflict, although it may at its discretion invoke help from other autonomic elements in doing so. Note that the element issuing the directive may expect, but not assume, that the directive will be carried out.

2.2 Recommended behaviors

The following behaviors are strongly encouraged, though not required, in autonomic elements.

An autonomic element *should* ask for a realistic set of requirements when requesting a service from another element. It should not, for instance, request a terabyte of storage when it knows it only needs a megabyte.

An autonomic element *should* offer a range of performance, reliability, availability and security associated with its service. This enables end-to-end optimization of these qualities in a system.

An autonomic element *should* be able to translate requirements for its service characteristics (performance, etc.) into requirements for any services that it needs to request from other elements. This enables self-assembly of systems without requiring central planning.

Finally, an autonomic element *should* protect itself against inappropriate service requests and responses. Specifically, it *should* authenticate all requests and requestors, and it *should* protect itself against inappropriate responses to its requests, e.g. by checking that each such response conforms to the pertinent agreement.

2.3 Policies

Of central importance to autonomic system behavior is the ability for high-level, broadly-scoped directives to be translated into specific actions to be taken by elements. This is achieved by the use of **policies**.

A policy is a representation, in a standard external form, of desired behaviors or constraints on behavior. Policy-based management of computer systems has been an active research topic for over a decade [7, 8]. For autonomic computing, the focus is specifically on policy-based *self*-management.

To cover the broadest possible range of situations and contexts, we allow for at least three interrelated forms of policy [9]. At the lowest level of specification are **action policies**, which are typically of the form IF (Condition) THEN (Action), e.g. IF (ResponseTime > 2 sec) THEN (Increase CPU share by 5%). An autonomic element employing action policies *must*

measure and/or synthesize the quantities stated in the condition, and it *must* execute the stated actions whenever the condition is satisfied.

At the next level are **goal policies**, which describe the conditions to be attained without specifying how to attain them, e.g. "Response time must not exceed 2 sec." Goal policies are more powerful than action policies because a human or an autonomic element can give direction to another element without requiring detailed knowledge of that element's inner workings. Autonomic elements employing goal policies *must* possess sufficient modeling or planning capabilities to translate goals into actions.

At the highest level are **utility function policies**, which specify the relative desirability of alternative states. This is achieved either by assigning a numerical value [10] or a partial or total ordering to the possible states [11]. Utility functions are even more powerful than goal policies because they automatically determine the most valuable goal in any given situation. Autonomic elements employing utility function policies *must* have sufficiently sophisticated modeling and optimization capabilities to translate utility functions into actions.

3. Autonomic element interactions

Having described the behaviors that we demand of autonomic elements, we now turn to how those elements interact with each other in a larger system. We describe the interfaces that autonomic elements must implement, the way in which they make agreements about what they will do for other elements, and restrictions on the ways in which they interact.

3.1 Interfaces

Any service-oriented architecture defines a number of standard interfaces through which services are described, discovered, and supplied. In order to achieve self-management and interoperability in autonomic systems, autonomic elements must implement additional interfaces as well. While space does not permit a detailed exposition of these interfaces here, we will briefly describe the major classes. In our current work, we define these interfaces as extensions of the OGSA architecture described in [4], but the concepts apply to any service-oriented architecture.

Monitoring and test interfaces enable an element to be monitored by any other element that has established the appropriate administrative relationship with it. These interfaces can be used to control the amount of logging and tracing data that an element gathers about its own operation, to gain access to that data, and in some cases to arrange for real-time feeds of the data. Related interfaces can be used to instruct an element to conduct a self-test, and to obtain the results of such tests.

Lifecycle interfaces enable administrative elements to determine the lifecycle state of an element (e.g. *starting*, *paused*), to cause that state to change (e.g. *shut down*), and to determine the lifecycle model that applies to the element.

Policy interfaces enable administrative elements to send new policies to an element, and to determine the policies currently in use by the element. The ability to send an element a new policy is not all-or-nothing; some elements may have a limited administrative relationship with the element (allowing, for instance, only new monitoring or alert policies to be accepted), while others may have total control, being authorized to replace or add any policy that the element understands.

Negotiation and binding interfaces permit an element to request a service from another element, or to be requested to provide a service. Simple forms of these interfaces allow an element to request a particular service, and receive either a confirmation or an error; this sort of interface is common to all service-oriented architectures. In order to achieve more flexible self-management, autonomic elements may also support more complex interfaces that allow proposals and counterproposals, negotiation over the exact terms and properties of the service to be provided (including levels of reliability, availability, performance, etc.), as well as allowing the formation and management of longer-term relationships.

Additional interfaces are discussed in Section 5.

3.2 Relationships

When an autonomic element has agreed to provide service to another autonomic element, we say that these two elements have a **relationship**. Typically, relationships are formed at run time rather than baked in during system deployment, and they may change over time. They are put into place by the elements themselves, rather than by human administrators.

Relationships are the way in which autonomic elements are composed to form autonomic systems. Indeed, in our approach, they are the only way in which elements are composed into larger entities.

In general, relationships are formed as a result of negotiation among the elements involved in them. An element will request the services of another element.

That request may be turned down (e.g. if the requesting element is not authorized, or if the requested element does not have sufficient resources). The requested element may counter propose a different relationship that it can accept (e.g. longer latency for transactions, but greater throughput). Once both elements agree on the terms of the relationship, they are bound by them and must seek to operate according to them.

This negotiation need not be laborious. It can be trivial, as in the case where the requestor is not authorized and is simply turned down. Similarly, some services may be implicitly available to all requesters, as is likely to be the case for the service interface used to ask an autonomic element for service in the first place.

3.3 Interaction integrity

Interaction integrity is a statement that an autonomic element is in control of all of its interactions with other elements. An autonomic element *must* communicate with other autonomic elements via service interfaces defined in its associated service specification. It *must not* communicate with other autonomic elements via any other mechanism (that is, there must not be any back channels). Communications within a given autonomic element must never be accessible outside the element in any way, e.g. as public service interfaces, via RMI or Java messaging, or in such a way as to violate other security assumptions, e.g. over an unauthenticated protocol.

This is a step beyond basic Web services and Grid services. Interaction integrity constrains how a service may be implemented, by limiting the ways in which an autonomic element can interact with, and be affected by, other autonomic elements: elements may interact only through their specified interfaces, and in no other way.

Interaction integrity enables an autonomic element to control its own behavior, since no other entity may "reach inside" and manipulate it directly. This is critical to self-protection; an autonomic element must be able to control what happens to it in a system. It is also vital to the ability of an autonomic element to make agreements that it knows it can fulfill, and in general to be able to manage its own behavior reliably.

4. Autonomic systems

Clearly, throwing self-managing components together arbitrarily does not guarantee self-management at the system level. For a system to function properly, its constituent elements must be able to discover each other, to identify other elements with which to communicate and to coordinate with each other in achieving their mutual goals. In addition, there are system-level behaviors that by their very nature cannot be performed by any single element, such as meeting end-to-end service level targets.

Suppose we have collected the autonomic elements that we need in order to implement a particular system, say a financial transaction system. We have router and firewall and Web server and database and storage elements. How do we assemble these into an *autonomic* financial transaction system?

Assembling an autonomic system requires:

1. A collection of autonomic elements that implement the desired function;
2. Additional autonomic elements to implement system functions that enable the needed system-level behaviors;
3. Design patterns for system self-management.

We already have the first of these in hand. The second entails the creation of a number of **infrastructure elements**—elements that support the operation of the autonomic system [4]. Some of these are briefly described below.

A **registry** provides mechanisms for elements to find one another, to publish their ability to perform certain types of service, and to determine how to bind to one another. When an element wants to find an element of a certain type, it first contacts a registry with which it has a relationship. It asks the registry about elements of that type. The registry returns a list of addresses to such elements. The first element can now contact each of the elements on that list, determine their suitability, form a relationship with the one it deems best suited to its needs, and use its service.

A **sentinel** provides monitoring services to other elements. See [12] for further discussion of sentinels.

An **aggregator** combines two or more existing elements and uses them to provide improved service—for example, an aggregator may be able to provide higher reliability or higher performance than any of the underlying elements could provide individually.

A **broker** facilitates interaction—it can, for example, assist an element in carrying out tasks requiring complex relationships of a sort that the element is not capable of entering into directly. Such an element would contact a broker and express its needs (e.g. a high availability storage service). The broker could create an aggregation of elements that fulfills this need (e.g. by composing underlying storage services) and return the address of the aggregate service to the requesting element.

A **negotiator** is an element that specializes in assisting elements with complex negotiations—for example, negotiations requiring a level of reasoning of which the client element is not capable. An element may know that it can trade off latency for throughput in a service that it needs to use, but not know the best protocol for exploring that trade-off. It can contact a negotiator, which can do that exploration for it, returning the best trade-off to the requesting element.

The third requirement for autonomic systems—design patterns—is the subject of the next section.

5. Design patterns

Here, we discuss initial patterns that we have used to institute self-configuration, self-healing, self-optimization and self-protection at a system level. This is not an exhaustive list; we expect that the invention of autonomic design patterns will be a fruitful area for quite some time.

5.1 Self-configuration

Self-configuration is an important part of the autonomic computing vision. Autonomic elements configure themselves, based on the environment in which they find themselves and the high-level tasks to which they have been set, without any detailed human intervention in the form of configuration files or installation dialogs.

It is possible to construct an autonomic system in much the same manner as we construct systems today. We could figure out all of the various dependencies and relationships at design time and instruct the individual elements to form predetermined relationships.

We have explored an alternative whereby the system builds itself, using what we call "goal-driven self-assembly" [12]. This potentially makes the system more robust, because system configuration decisions are made locally. Before each element joins the system, it is given a high-level description of what it is supposed to be doing ("make yourself available as an application server", or "join policy repository cluster 17"), and how to contact the registry.

When each element initializes, it contacts the registry and issues queries to locate existing elements that can supply the services that the new element needs to operate. It contacts the elements thus located, and enters into relationships as required to obtain the needed services. Once the element has entered into all the relationships and obtained all the resources that it needs to function, it registers itself in the registry so that elements that later need its services can contact it.

Once all of the elements have satisfied their goals, the system as a whole has self-assembled.

There has been considerable previous work in the fields of service discovery (see [13] and [14]) and service registries (see [15] and [16]). These concepts could be modernized to support autonomic computing by translating them into a service-oriented architectural paradigm and supplementing them with appropriate service ontologies.

5.2 Self-healing

No matter how robust and resilient they are, we recognize that autonomic elements will still fail from time to time. We demand that the robustness and resiliency of an autonomic computing system not depend on the robustness and resiliency of any single autonomic element. That is, the system as a whole should be capable of dealing with the failure of any constituent part.

We seek to ensure this property at an architectural level. Designers may, in some circumstances, be able to deal with failure by using techniques that are idiosyncratic to the particular element that they are designing. A storage system may have special hardware that copies data off of failing disks. A weather prediction module may be able to check that its results are physically consistent. We want something more. We want no *architectural* single point of failure. That is, it should be possible, within the architecture, for the system as a whole to be self-healing.

To do this, we make each autonomic element responsible for monitoring its input services and determining if those services are performing according to the negotiated agreement covering them. If the input service fails, whether entirely, or because its performance is out of bounds, or because the results it returns are incorrect, the requesting autonomic element will react, possibly by terminate its relationship with the input service and obtaining a new one.

If the input service is stateless, this is relatively straightforward. A new input service can be found via the registry. In the case where not having an input service for a period of time would cause a problem, the requesting element should arrange for a standby for its input service, ensuring that it is provisioned and ready to go ahead of time. Then, should an input service fail, the requesting service can switch over to the standby input service quickly. Alternatively, clusters of input services can be created ahead of time, and requests handed out to them via round-robin mechanisms.

Should one of the autonomic elements in the cluster fail, the others in the cluster can absorb the load.

If the input service is stateful, the state of the input service must also be resilient against failure. There is a substantial literature on this topic in the field of distributed and fault-tolerant computing, particularly as it relates to loosely-coupled distributed Web services [17, 18]. We illustrate one simple method here.

The requesting service can maintain two mirror images of the (stateful) input service, sending identical transactions to both of them in order to keep them in synch. If one input service fails, the requesting service can first temporarily suspend its transactions with the second input service, find (or generate) a new input service of the appropriate type, and populate the new input service by copying state from the old input service (the one that did not fail) to the new input service. When the new input service is brought up to the same state as the old input service, the requesting service can return to mirroring transactions to the two input services.

If this re-provisioning takes a long time, suspending the service will be impractical, and other ways of provisioning a new input service will be needed. We expect these to be idiosyncratic to both the requesting service and the input service.

There are several places in which this self-healing functionality may be located. It may be in the requesting element, as in our example. It may be in the input service, which makes the input service internally self-healing. It may be in an intermediary, aggregating less-reliable input services into a more reliable aggregate service. Each has its advantages and disadvantages, but no one choice is sufficient by itself.

One useful design pattern for self-healing, which avoids the problem of having a single point of failure, is the **self-regenerating cluster**. The concept is to cluster two or more instances of a particular type of autonomic element together, such that they share input services and respond to requests for output services via round robin or spraying techniques. The autonomic elements in such a cluster could monitor each other's health. If one such autonomic element were to fail, one of the remaining autonomic elements could generate (or find) a new instance of that type, bind it into the cluster, and thus reconstitute the cluster.

To support this self-healing functionality, an autonomic element *should* expose interfaces which enable the following:

Sending state. This is a request sent to an element, instructing it to send its internal state to another element—the message specifies the receiving element.

Receiving state. This is a request sent to an element, instructing it to receive internal state from another such element—the message specifies the sending element.

The state exchange could be initiated by a third element—an administrative element—or it could be initiated by either the sender or the receiver.

To support end-to-end availability management, an autonomic element *should* expose interfaces which enable the following:

Querying planned outages. This is a request sent to the element, to which it replies with a list of planned outages in a specified period. The outages may be partial—i.e., the element may be unable to provide services to other element, but may still be able to respond to simple queries; or they may be total—i.e., the element may be unable to respond at all.

Scheduling planned outages. This is a message sent to an element, specifying a time period in which the element may safely initiate a planned outage—e.g., to reboot, or to perform internal operations that cannot be done without compromising service levels.

5.3 Self-optimization

Self-optimization at the system level is obviously related to self-optimization of the individual components. But good behavior of each component does not necessarily ensure good behavior of the system as a whole. Furthermore, in any system, it is likely that conflicts will have to be resolved, such as when two components both want control over a limited resource.

One design pattern for resource allocation that has received some attention has been the use of market-like mechanisms [19]. This has the advantage of being very general, but does incur additional real cost, both in run time efficiency and code complexity; and to be effective, the convergence time of the market's prices must be smaller than the required response time of the system. For this pattern to be used, both the "buyer" and "seller" elements must be able to correctly determine the value of the services in question. This may be done, e.g., by enabling the elements with utility functions that assign values to the different services they might buy or sell.

Another pattern, found in one of the prototypes described in Section 6 below, has a resource-arbiter element that directly queries elements' utility functions, and then combines them to calculate a system-wide optimal allocation.

Each of these patterns places its own requirements on the elements involved. Both, however, presume that an element will be able to offer different levels of service. Therefore, autonomic elements *should* offer multiple levels of service.

Service levels may be offered per individual relationship, so that each relationship has its own set of service goals, or even per work item, so that each job is assigned a service goal. It is often convenient to define service classes, each of which has its own set of service goals.

Elements *may* offer multiple levels of service by participating in parallel clusters. Thus, for example, a single element may not be able to provide a given service level on its own, but it may be able to form relationships with other elements to achieve the desired service collectively. This pattern obviously applies only to work that can be parallelized.

The ability to offer multiple levels of service may require an element to provide interfaces that include:

Querying service-level bounds. This is a request sent to the element, to which it replies with its current service goals and constraints. The request may identify a particular service class, relationship or work item.

Querying service level. This is a request sent to the element, to which it replies with the current service level. The request may identify a particular service class, relationship, or work item.

Requesting a service level. This is a request to modify the current service levels.

5.4 Self-protection

There are two distinct but related aspects to self-protection: protection against undesirable system behavior due to bugs or unanticipated conditions, and protection against system penetration by attackers [20]. Many of the principles and design patterns described elsewhere in this paper are effective against accidental conditions. The ability of a system to dynamically self-optimize protects the system's performance against changes in demand, for instance, and self-healing functions protect a system against degradation due to the accumulation of failures over time.

To the extent that autonomic functions protect against accidental failures, they will also protect against some types of maliciously-induced failure. If an autonomic self-healing system can quickly and transparently replace a Web server when it crashes due to a hardware failure, that same system can recover from a crash caused by an attacker intentionally exploiting a network software bug to take down the machine.

Other kinds of malicious attack, on the other hand, require special handling. Accidental failures tend to be uncorrelated; failures due to malice, can be highly correlated, as when an attacker stages a denial of service attack against many systems at once, or when one machine is attacked to draw attention away from another which is the real target. Autonomic intrusion defense systems that can detect and respond to these correlated failures in real time may share infrastructure with, but will often use different rules than, event correlators that detect accidental system failure.

A key principle of autonomic computing is that the system as a whole should continue to function even if one or more of its elements fails. In the security realm, this corresponds to a system that continues to be secure as a whole even when one or more of its elements is compromised. This is a challenging goal, towards which some progress is being made. (See, for instance, the secure distributed storage system described in [21].) Similar design patterns will be needed in other aspects of autonomic systems, particularly security-critical ones. For instance, particularly sensitive operations might be authorized only if multiple autonomic elements all register assent, significantly raising the bar for an attacker.

Like any other computing system, autonomic systems will require access controls and other traditional security controls, dictating not only which users are authorized to take which actions, but also which autonomic elements are authorized. In autonomic systems, we anticipate that these security controls will be implemented through policies. Security policies in autonomic systems will benefit from the entire general policy infrastructure used in autonomic systems. It will be possible to deploy security policies into common policy repositories, it will be possible to detect conflicts between policies, both at policy creation time and at run time, and it will be possible to use advanced policy tools to explore the consequences of proposed policy changes before they are implemented. Standardizing security controls in the form of autonomic policies should reduce the complexity and confusion that currently exists, by reducing the number of incompatible and non-interoperable security management mechanisms, and leveraging common tooling and infrastructure.

6. Autonomic computing system prototypes

We have validated and refined these architectural ideas by creating two prototype autonomic systems [12, 22] that explore the use of autonomic systems for data center management and resource allocation. Autonomic elements represent applications running in the data center, the resources that those applications require to run, and the resource arbiter that allocates the resources to the applications. Other autonomic elements provide many of the infrastructural services described above, including a registry and a policy

repository. Goal-driven self-assembly and utility-function policies are key elements of one of these prototype systems, which uses some of the design patterns described above to achieve a degree of self-management.

7. Conclusion

We have described an architectural approach for creating autonomic elements (self-managing components), and for composing them to form autonomic systems (self-managing systems).

Our approach takes advantage of the uniform representation and composition of components in service-oriented architectures and the autonomy of components in agent-oriented programming. It goes beyond previous work by deriving component-level self-management from the interfaces and behaviors that it requires of autonomic elements. It derives system-level self-management by composing autonomic elements via negotiated relationships, adding to them infrastructure elements (registries, service brokers, etc.), and requiring that they follow a growing list of system design patterns that we have developed.

There is much to be done. We (and others) are working actively on detailed interface specifications, and reference implementations of them. We are working through the details of the design patterns described here and validating them with a more comprehensive prototype.

8. References

[1] W. Asprey, et al., "Conquer System Complexity: Build Systems with Billions of Parts," in *CRA Conference on Grand Research Challenges in Computer Science and Engineering*, Warrenton, VA (2002), pp. 29-33, http://www.cra.org/reports/gc.systems.pdf.

[2] J. Kephart and D. Chess, "The Vision of Autonomic Computing," *IEEE Computer*, Vol. 36, No. 1 (2003), pp. 41-50, http://www.research.ibm.com/autonomic/research/papers/AC_Vision_Computer_Jan_2003.pdf.

[3] P. Horn, "Autonomic Computing: IBM's Perspective on the State of Information Technology," IBM Corp. (October 2001), http://www.research.ibm.com/autonomic/manifesto/autonomic_computing.pdf.

[4] I. Foster, C. Kesselman and St. Tuecke, "The Anatomy of the Grid: Enabling Scalable Virtual Organizations" (2001), http://www.globus.org/research/papers/anatomy.pdf .

[5] M. Champion, C. Ferris, E. Newcomer, and D. Orchard, "Web Services Architecture," W3C Working Draft, (November 14, 2002), http://www.w3.org/TR/ws-arch/.

[6] M. Wooldridge and N. Jennings, "Agent theories, architectures, and languages," in Wooldridge and Jennings (eds.), *Intelligent Agents*, Springer-Verlag (1995), pp. 1-22.

[7] M. Sloman, "Policy Driven Management for Distributed Systems", *Journal of Network and Systems Management*, Vol. 2 (1994).

[8] Policy Workshop: International Workshop on Policies for Distributed Systems and Networks, http://www.policy-workshop.org/.

[9] J. Kephart, W. Walsh, "An Artificial Intelligence Perspective on Autonomic Computing Policies," *Proceedings of Policy 2004*, to be published.

[10] W. Walsh, G. Tesauro, J. Kephart and R. Das, "Utility Functions in Autonomic Systems," *Proceedings of First International Conference on Autonomic Computing* (2004).

[11] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, Prentice Hall (2003), second edition.

[12] D. Chess, A. Segal, I. Whalley and S. White, "Unity: Experiences with a Prototype Autonomic Computing System," *Proceedings of First International Conference on Autonomic Computing* (2004).

[13] Jini.org, "Jini Core Technology Spec – Discovery and Join," http://www.jini.org/nonav/standards/davis/doc/specs/html/discovery-spec.html.

[14] UPnP forum, "UPnP Device Architecture 1.0," v1.0.1, 2 December 2003, http://www.upnp.org/resources/documents/CleanUPnPDA101-20031202s.pdf.

[15] Java Naming and Directory Interface, http://java.sun.com/products/jndi/reference/docs/index.html.

[16] UDDI.org, "Universal Description, Discovery and Integration v3.0.1," http://uddi.org/pubs/uddi_v3.htm.

[17] J. Roberts, K. Srinivasan; "Tentative Hold Protocol Part 1: White Paper," *W3C Note* (November 28, 2001), http://www.w3.org/TR/tenthold-1/.

[18] T. Mikalsen, St. Tai and I. Rouvellou; "Transactional Attitudes: Reliable Composition of Autonomous Web Services," *Workshop on Dependable Middleware-based Systems (WDMS 2002)*, June 2002, http://www.research.ibm.com/AEM/pubs/wstx-WDMS-DSN2002.pdf.

[19] S. Clearwater (ed.), *Market-Based Control,* World Scientific, Singapore (1996).

[20] D. Chess, C. Palmer, and S. White, "Security in an autonomic computing environment," *IBM Systems Journal*, Vol. 42, No. 1 (2003).

[21] J. Garay, R. Gennaro, C. Jutla and T. Rabin, "Secure Distributed Storage and Retrieval," *Theoretical Computer Science*, Vol. 243, No. 1–2 (2000), pp. 363–389.

[22] M. Devarakonda, D. Chess, I. Whalley, A. Segal, P. Goyal, A. Sachedina, K. Romanufa, E. Lassettre, W. Tetzlaff, B. Arnold; "Policy-based, Autonomic Storage Allocation," *Proc. of 14th IFIP/IEEE Intl. Workshop on Distributed Systems: Operations and Management, DSOM 2003*, M. Brunner and A, Keller (Eds.), *Lecture Notes on Computer Science 2867*, Springer-Verlag