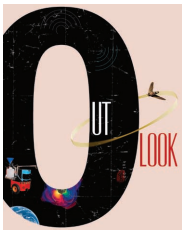
The Vision of Autonomic Computing



Systems manage themselves according to an administrator's goals. New components integrate as effortlessly as a new cell establishes itself in the human body. These ideas are not science fiction, but elements of the grand challenge to create self-managing computing systems.

Jeffrey O. Kephart
David M. Chess
IBM Thomas J. Watson Research Center

In mid-October 2001, IBM released a manifesto observing that the main obstacle to further progress in the IT industry is a looming software complexity crisis.[1] The company cited applications and environments that weigh in at tens of millions of lines of code and require skilled IT professionals to install, configure, tune, and maintain.

The manifesto pointed out that the difficulty of managing today's computing systems goes well beyond the administration of individual software environments. The need to integrate several heterogeneous environments into corporate-wide computing systems, and to extend that beyond company boundaries into the Internet, introduces new levels of complexity. Computing systems' complexity appears to be approaching the limits of human capability, yet the march toward increased interconnectivity and integration rushes ahead unabated.

This march could turn the dream of pervasive computing—trillions of computing devices connected to the Internet—into a nightmare. Programming language innovations have extended the size and complexity of systems that architects can design, but relying solely on further innovations in programming methods will not get us through the present complexity crisis.

As systems become more interconnected and diverse, architects are less able to anticipate and design interactions among components, leaving such issues to be dealt with at runtime. Soon systems will become too massive and complex for even the most skilled system integrators to install, configure, optimize, maintain, and merge. And there will be no way to make timely, decisive responses to the rapid stream of changing and conflicting demands.

AUTONOMIC OPTION

The only option remaining is *autonomic computing*—computing systems that can manage themselves given high-level objectives from administrators. When IBM's senior vice president of research, Paul Horn, introduced this idea to the National Academy of Engineers at Harvard University in a March 2001 keynote address, he deliberately chose a term with a biological connotation. The autonomic nervous system governs our heart rate and body temperature, thus freeing our conscious brain from the burden of dealing with these and many other low-level, yet vital, functions.

The term autonomic computing is emblematic of a vast and somewhat tangled hierarchy of natural self-governing systems, many of which consist of myriad interacting, self-governing components that in turn comprise large numbers of interacting, autonomous, self-governing components at the next level down. The enormous range in scale, starting with molecular machines within cells and extending to human markets, societies, and the entire world socioeconomy, mirrors that of computing systems, which run from individual devices to the entire Internet. Thus, we believe it will be profitable to seek inspiration in the self-governance of social and economic systems as well as purely biological ones.

Clearly then, autonomic computing is a grand

0018-9162/03/$17.00 © 2003 IEEE

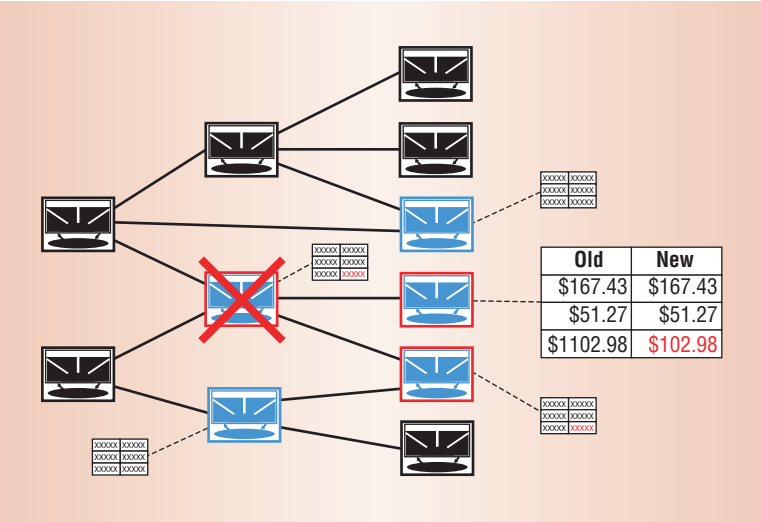


Figure 1. Problem diagnosis in an autonomic system upgrade. The upgrade introduces five software modules (blue), each an autonomic element. Minutes after installation, regression testers find faulty output in three of the new modules (red outlines), and the system immediately reverts to its old version. A problem determiner, an autonomic element, obtains information about interelement dependencies (lines between elements) from a dependency analyzer, another autonomic element that probes the system periodically (not shown). Taking into account its knowledge of interelement dependencies, the problem determiner analyzes log files and infers which of the three potentially bad modules is the culprit (red X). It generates a problem ticket containing diagnostic information and sends it to a software developer, who debugs the module and makes it available for future upgrades.

challenge that reaches far beyond a single organization. Its realization will take a concerted, long-term, worldwide effort by researchers in a diversity of fields. A necessary first step is to examine this vision: what autonomic computing systems might look like, how they might function, and what obstacles researchers will face in designing them and understanding their behavior.

SELF-MANAGEMENT

The essence of autonomic computing systems is self-management, the intent of which is to free system administrators from the details of system operation and maintenance and to provide users with a machine that runs at peak performance 24/7.

Like their biological namesakes, autonomic systems will maintain and adjust their operation in the face of changing components, workloads, demands, and external conditions and in the face of hardware or software failures, both innocent and malicious. The autonomic system might continually monitor its own use, and check for component upgrades, for example. If it deems the advertised features of the upgrades worthwhile, the system will install them, reconfigure itself as necessary, and run a regression test to make sure all is well. When it detects errors, the system will revert to the older version while its automatic problem-determination algorithms try to isolate the source of the error. Figure 1 illustrates how this process might work for an autonomic accounting system upgrade.

IBM frequently cites four aspects of self-management, which Table 1 summarizes. Early autonomic systems may treat these aspects as distinct, with different product teams creating solutions that address each one separately. Ultimately, these aspects will be emergent properties of a general architecture, and distinctions will blur into a more general notion of self-maintenance.

The journey toward fully autonomic computing will take many years, but there are several important and valuable milestones along the path. At first, automated functions will merely collect and aggregate information to support decisions by human administrators. Later, they will serve as advisors, suggesting possible courses of action for humans to consider. As automation technologies improve, and our faith in them grows, we will entrust autonomic systems with making—and acting on—lower-level decisions. Over time, humans will need to make relatively less frequent predominantly higher-level decisions, which the system will carry out automatically via more numerous, lower-level decisions and actions.

Ultimately, system administrators and end users will take the benefits of autonomic computing for granted. Self-managing systems and devices will seem completely natural and unremarkable, as will automated software and middleware upgrades. The detailed migration patterns of applications or data will be as uninteresting to us as the details of routing a phone call through the telephone network.

Self-configuration

Installing, configuring, and integrating large, complex systems is challenging, time-consuming, and error-prone even for experts. Most large Web sites and corporate data centers are haphazard accretions of servers, routers, databases, and other technologies on different platforms from different vendors. It can take teams of expert programmers months to merge two systems or to install a major e-commerce application such as SAP.

Autonomic systems will configure themselves automatically in accordance with high-level policies—representing business-level objectives, for

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

Concept	Current computing	Autonomic computing
Self-configuration	Corporate data centers have multiple vendors and platforms. Installing, configuring, and integrating systems is time consuming and error prone.	Automated configuration of components and systems follows high-level policies. Rest of system adjusts automatically and seamlessly.
Self-optimization	Systems have hundreds of manually set, nonlinear tuning parameters, and their number increases with each release.	Components and systems continually seek opportunities to improve their own performance and efficiency.
Self-healing	Problem determination in large, complex systems can take a team of programmers weeks.	System automatically detects, diagnoses, and repairs localized software and hardware problems.
Self-protection	Detection of and recovery from attacks and cascading failures is manual.	System automatically defends against malicious attacks or cascading failures. It uses early warning to anticipate and prevent systemwide failures.

example—that specify what is desired, not how it is to be accomplished. When a component is introduced, it will incorporate itself seamlessly, and the rest of the system will adapt to its presence—much like a new cell in the body or a new person in a population. For example, when a new component is introduced into an autonomic accounting system, as in Figure 1, it will automatically learn about and take into account the composition and configuration of the system. It will register itself and its capabilities so that other components can either use it or modify their own behavior appropriately.

Self-optimization

Complex middleware, such as WebSphere, or database systems, such as Oracle or DB2, may have hundreds of tunable parameters that must be set correctly for the system to perform optimally, yet few people know how to tune them. Such systems are often integrated with other, equally complex systems. Consequently, performance-tuning one large subsystem can have unanticipated effects on the entire system.

Autonomic systems will continually seek ways to improve their operation, identifying and seizing opportunities to make themselves more efficient in performance or cost. Just as muscles become stronger through exercise, and the brain modifies its circuitry during learning, autonomic systems will monitor, experiment with, and tune their own parameters and will learn to make appropriate choices about keeping functions or outsourcing them. They will proactively seek to upgrade their function by finding, verifying, and applying the latest updates.

Self-healing

IBM and other IT vendors have large departments devoted to identifying, tracing, and determining the root cause of failures in complex computing systems. Serious customer problems can take teams of programmers several weeks to diagnose and fix, and sometimes the problem disappears mysteriously without any satisfactory diagnosis.

Autonomic computing systems will detect, diagnose, and repair localized problems resulting from bugs or failures in software and hardware, perhaps through a regression tester, as in Figure 1. Using knowledge about the system configuration, a problem-diagnosis component (based on a Bayesian network, for example) would analyze information from log files, possibly supplemented with data from additional monitors that it has requested. The system would then match the diagnosis against known software patches (or alert a human programmer if there are none), install the appropriate patch, and retest.

Self-protection

Despite the existence of firewalls and intrusion-detection tools, humans must at present decide how to protect systems from malicious attacks and inadvertent cascading failures.

Autonomic systems will be self-protecting in two senses. They will defend the system as a whole against large-scale, correlated problems arising from malicious attacks or cascading failures that remain uncorrected by self-healing measures. They also will anticipate problems based on early reports from sensors and take steps to avoid or mitigate them.

ARCHITECTURAL CONSIDERATIONS

Autonomic systems will be interactive collections of *autonomic elements*—individual system constituents that contain resources and deliver services to humans and other autonomic elements. Autonomic elements will manage their internal behavior and their relationships with other autonomic elements in accordance with policies that humans or other elements have established. *System* self-management will arise at least as much from the myriad

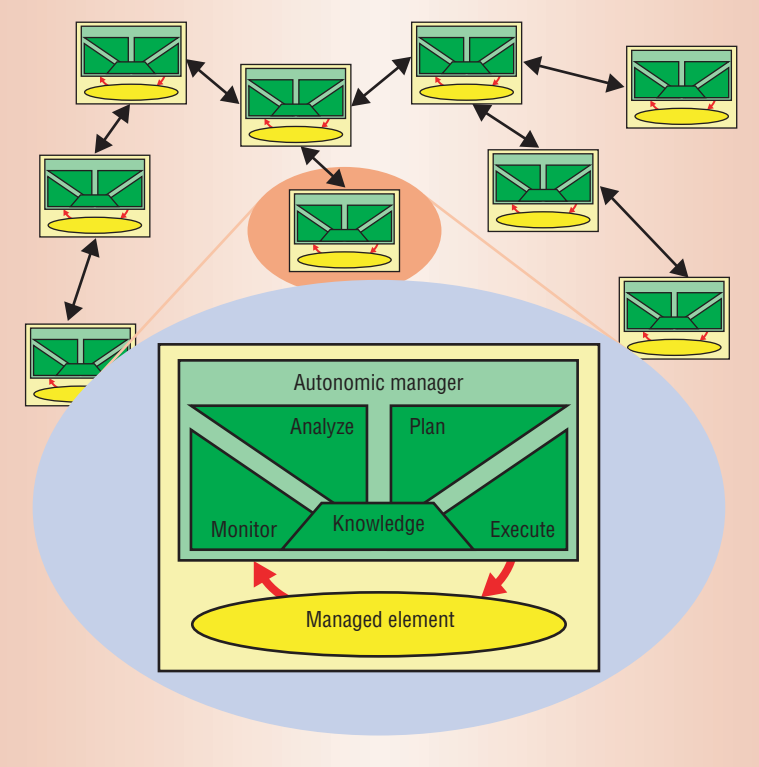


Figure 2. Structure of an autonomic element. Elements interact with other elements and with human programmers via their autonomic managers.

interactions among autonomic elements as it will from the internal self-management of the individual autonomic elements—just as the social intelligence of an ant colony arises largely from the interactions among individual ants. A distributed, service-oriented infrastructure will support autonomic elements and their interactions.

As Figure 2 shows, an autonomic element will typically consist of one or more managed elements coupled with a single autonomic manager that controls and represents them. The managed element will essentially be equivalent to what is found in ordinary nonautonomic systems, although it can be adapted to enable the autonomic manager to monitor and control it. The managed element could be a hardware resource, such as storage, a CPU, or a printer, or a software resource, such as a database, a directory service, or a large legacy system.

At the highest level, the managed element could be an e-utility, an application service, or even an individual business. The autonomic manager distinguishes the autonomic element from its nonautonomic counterpart. By monitoring the managed element and its external environment, and constructing and executing plans based on an analysis of this information, the autonomic manager will relieve humans of the responsibility of directly managing the managed element.

Fully autonomic computing is likely to evolve as designers gradually add increasingly sophisticated autonomic managers to existing managed elements. Ultimately, the distinction between the autonomic manager and the managed element may become merely conceptual rather than architectural, or it may melt away—leaving fully integrated, autonomic elements with well-defined behaviors and interfaces, but also with few constraints on their internal structure.

Each autonomic element will be responsible for managing its own internal state and behavior and for managing its interactions with an environment that consists largely of signals and messages from other elements and the external world. An element's internal behavior and its relationships with other elements will be driven by goals that its designer has embedded in it, by other elements that have authority over it, or by subcontracts to peer elements with its tacit or explicit consent. The element may require assistance from other elements to achieve its goals. If so, it will be responsible for obtaining necessary resources from other elements and for dealing with exception cases, such as the failure of a required resource.

Autonomic elements will function at many levels, from individual computing components such as disk drives to small-scale computing systems such as workstations or servers to entire automated enterprises in the largest autonomic system of all—the global economy.

At the lower levels, an autonomic element's range of internal behaviors and relationships with other elements, and the set of elements with which it can interact, may be relatively limited and hard-coded. Particularly at the level of individual components, well-established techniques—many of which fall under the rubric of fault tolerance—have led to the development of elements that rarely fail, which is one important aspect of being autonomic. Decades of developing fault-tolerance techniques have produced such engineering feats as the IBM zSeries servers, which have a mean time to failure of several decades.

At the higher levels, fixed behaviors, connections, and relationships will give way to increased dynamism and flexibility. All these aspects of autonomic elements will be expressed in more high-level, goal-oriented terms, leaving the elements themselves with the responsibility for resolving the details on the fly.

Hard-coded behaviors will give way to behaviors expressed as high-level objectives, such as "maximize this utility function," or "find a reputable message translation service." Hardwired connections among elements will give way to increasingly less direct specifications of an element's partners—from specification by physical address to specification by name and finally to specification by function, with the partner's identity being resolved only when it is needed.

Hard-wired relationships will evolve into flexible relationships that are established via negotiation. Elements will automatically handle new modes of failure, such as contract violation by a supplier, without human intervention.

While service-oriented architectural concepts like Web and grid services[2,3] will play a fundamental role, a sufficient foundation for autonomic computing requires more. First, as service providers, autonomic elements will not unquestioningly honor requests for service, as would typical Web services or objects in an object-oriented environment. They will provide a service only if providing it is consistent with their goals. Second, as consumers, autonomic elements will autonomously and proactively issue requests to other elements to carry out their objectives.

Finally, autonomic elements will have complex life cycles, continually carrying on multiple threads of activity, and continually sensing and responding to the environment in which they are situated. Autonomy, proactivity, and goal-directed interactivity with their environment are distinguishing characteristics of software agents. Viewing autonomic elements as agents and autonomic systems as multiagent systems makes it clear that agent-oriented architectural concepts will be critically important.[4]

ENGINEERING CHALLENGES

Virtually every aspect of autonomic computing offers significant engineering challenges. The life cycle of an individual autonomic element or of a relationship among autonomic elements reveals several challenges. Others arise in the context of the system as a whole, and still more become apparent at the interface between humans and autonomic systems.

Life cycle of an autonomic element

An autonomic element's life cycle begins with its design and implementation; continues with test and verification; proceeds to installation, configuration, optimization, upgrading, monitoring, problem determination, and recovery; and culminates in uninstallation or replacement. Each of these stages has special issues and challenges.

Design, test, and verification. Programming an autonomic element will mean extending Web services or grid services with programming tools and techniques that aid in managing relationships with other autonomic elements. Because autonomic elements both consume and provide services, representing needs and preferences will be just as important as representing capabilities. Programmers will need tools that help them acquire and represent policies—high-level specifications of goals and constraints, typically represented as rules or utility functions—and map them onto lower-level actions. They will also need tools to build elements that can establish, monitor, and enforce agreements.

Autonomic elements will provide a service only if it is consistent with their goals.

Testing autonomic elements and verifying that they behave correctly will be particularly challenging in large-scale systems because it will be harder to anticipate their environment, especially when it extends across multiple administrative domains or enterprises. Testing networked applications that require coordinated interactions among several autonomic elements will be even more difficult.

It will be virtually impossible to build test systems that capture the size and complexity of realistic systems and workloads. It might be possible to test newly deployed autonomic elements in situ by having them perform alongside more established and trusted elements with similar functionality.

The element's potential customers may also want to test and verify its behavior, both before establishing a service agreement and while the service is provided. One approach is for the autonomic element to attach a testing method to its service description.

Installation and configuration. Installing and configuring autonomic elements will most likely entail a bootstrapping process that begins when the element registers itself in a directory service by publishing its capabilities and contact information. The element might also use the directory service to discover suppliers or brokers that may provide information or services it needs to complete its initial configuration. It can also use the service to seek out potential customers or brokers to which it can delegate the task of finding customers.

Monitoring and problem determination. Monitoring will be an essential feature of autonomic elements. Elements will continually monitor themselves to ensure that they are meeting their own objectives, and they will log this information to serve as the

The vision of autonomic systems as a complex supply web makes problem determination both easier and harder than it is now.

basis for adaptation, self-optimization, and reconfiguration. They will also continually monitor their suppliers to ensure that they are receiving the agreed-on level of service and their customers to ensure that they are not exceeding the agreed-on level of demand. Special sentinel elements may monitor other elements and issue alerts to interested parties when they fail.

When coupled with event correlation and other forms of analysis, monitoring will be important in supporting problem determination and recovery when a fault is found or suspected. Applying monitoring, audit, and verification tests at all the needed points without burdening systems with excessive bandwidth or processing demands will be a challenge. Technologies to allow statistical or sample-based testing in a dynamic environment may prove helpful.

The vision of autonomic systems as a complex supply web makes problem determination both easier and harder than it is now. An autonomic element that detects poor performance or failure in a supplier may not attempt a diagnosis; it may simply work around the problem by finding a new supplier.

In other situations, however, it will be necessary to determine why one or more elements are failing, preferably without shutting down and restarting the entire system. This requires theoretically grounded tools for tracing, simulation, and problem determination in complex dynamic environments. Particularly when autonomic elements—or applications based on interactions among multiple elements—have a large amount of state, recovering gracefully and quickly from failure or restarting applications after software has been upgraded or after a function has been relocated to new machines will be challenging. David Patterson and colleagues at the University of California, Berkeley, and Stanford University have made a promising start in this direction.[5]

Upgrading. Autonomic elements will need to upgrade themselves from time to time. They might subscribe to a service that alerts them to the availability of relevant upgrades and decide for themselves when to apply the upgrade, possibly with guidance from another element or a human. Alternatively, the system could create afresh entirely new elements as part of a system upgrade, eliminating outmoded elements only after the new ones establish that they are working properly.

Managing the life cycle. Autonomic elements will typically be engaged in many activities simultaneously: participating in one or more negotiations at various phases of completion, proactively seeking inputs from other elements, and so on. They will need to schedule and prioritize their myriad activities, and they will need to represent their life cycle so that they can both reason about it and communicate it to other elements.

Relationships among autonomic elements

In its most dynamic and elaborate form, the service relationship among autonomic elements will also have a life cycle. Each stage of this life cycle engenders its own set of engineering challenges and standardization requirements.

Specification. An autonomic element must have associated with it a set of output services it can perform and a set of input services that it requires, expressed in a standard format so that other autonomic elements can understand it. Typically, the element will register with a directory service such as Universal Description, Discovery, and Integration[6] or an Open Grid Services Architecture (OGSA) registry,[3] providing a description of its capabilities and details about addresses and the protocols other elements or people can use to communicate with it.

Establishing standard service ontologies and a standard service description syntax and semantics that are sufficiently expressive for machines to interpret and reason about is an area of active research. The US Defense Advanced Research Projects Agency's semantic Web effort[7] is representative.

Location. An autonomic element must be able to locate input services that it needs; in turn, other elements that require its output services must be able to locate that element.

To locate other elements dynamically, the element can look them up by name or function in a directory service, possibly using a search process that involves sophisticated reasoning about service ontologies. The element can then contact one or more potential service providers directly and converse with them to determine if it can provide exactly the service they require.

In many cases, autonomic elements will also need to judge the likely reliability or trustworthiness of potential partners—an area of active research with many unsolved fundamental problems.

Negotiation. Once an element finds potential providers of an input service, it must negotiate with them to obtain that service.

We construe negotiation broadly as any process by which an agreement is reached. In *demand-for-service* negotiation, the element providing a service is subservient to the one requesting it, and the

provider must furnish the service unless it does not have sufficient resources to do so. Another simple form of negotiation is *first-come, first-served*, in which the provider satisfies all requests until it runs into resource limitations. In *posted-price* negotiation, the provider sets a price in real or artificial currency for its service, and the requester must take it or leave it.

More complex forms of negotiation include bilateral or multilateral negotiations over multiple attributes, such as price, service level, and priority, involving multiple rounds of proposals and counterproposals. A third-party arbiter can run an auction or otherwise assist these more complex negotiations, especially when they are multilateral.

Negotiation will be a rich source of engineering and scientific challenges for autonomic computing. Elements need flexible ways to express multiattribute needs and capabilities, and they need mechanisms for deriving these expressions from human input or from computation. They also need effective negotiation strategies and protocols that establish the rules of negotiation and govern the flow of messages among the negotiators. There must be languages for expressing service agreements—the culmination of successful negotiation—in their transient and final forms.

Efforts to standardize the representation of agreements are under way, but mechanisms for negotiating, enforcing, and reasoning about agreements are lacking, as are methods for translating them into action plans.

Provision. Once two elements reach an agreement, they must provision their internal resources. Provision may be as simple as noting in an access list that a particular element can request service in the future, or it may entail establishing additional relationships with other elements, which become subcontractors in providing some part of the agreed-on service or task.

Operation. Once both sides are properly provisioned, they operate under the negotiated agreement. The service provider's autonomic manager oversees the operation of its managed element, monitoring it to ensure that the agreement is being honored; the service requester might similarly monitor the level of service.

If the agreement is violated, one or both elements would seek an appropriate remedy. The remedy may be to assess a penalty, renegotiate the agreement, take technical measures to minimize any harm from the failure, or even terminate the agreement.

Termination. When the agreement has run its course, the parties agree to terminate it, freeing their internal resources for other uses and terminating agreements for input services that are no longer needed. The parties may record pertinent information about the service relationship locally, or store it in a database a reputation element maintains.

Systemwide issues

Other important engineering issues that arise at the system level include security, privacy, and trust, and the emergence of new types of services to serve the needs of other autonomic elements.

System-level engineering issues include security, privacy, and trust, and new types of services to serve the needs of other autonomic elements.

Autonomic computing systems will be subject to all the security, privacy, and trust issues that traditional computing systems must now address. Autonomic elements and systems will need to both establish and abide by security policies, just as human administrators do today, and they will need to do so in an understandable and fail-safe manner.

Systems that span multiple administrative domains—especially those that cross company boundaries—will face many of the challenges that now confront electronic commerce. These include authentication, authorization, encryption, signing, secure auditing and monitoring, nonrepudiation, data aggregation and identity masking, and compliance with complex legal requirements that vary from state to state or country to country.

The autonomic systems infrastructure must let autonomic elements identify themselves, verify the identities of other entities with which they communicate, verify that a message has not been altered in transit, and ensure that unauthorized parties do not read messages and other data. To satisfy privacy policies and laws, elements must also appropriately protect private and personal information that comes into their possession. Measures that keep data segregated according to its origin or its purpose must be extended into the realm of autonomic elements to satisfy policy and legal requirements.

Autonomic systems must be robust against new and insidious forms of attack that use self-management based on high-level policies to their own advantage. By altering or otherwise manipulating high-level policies, an attacker could gain much greater leverage than is possible in nonautonomic systems. Preventing such problems may require a new subfield of computer security that seeks to thwart fraud and the fraudulent persuasion of autonomic elements.

On a larger scale, autonomic elements will be

To satisfy privacy policies and laws, elements must appropriately protect information that comes into their possession.

agents, and autonomic systems will in effect be multiagent systems built on a Web services or OGSA infrastructure. Autonomic systems will be inhabited by middle agents[8] that serve as intermediaries of various types, including directory services, matchmakers, brokers, auctioneers, data aggregators, dependency managers—for detecting, recording, and publicizing information about functional dependencies among autonomic elements—event correlators, security analysts, time-stampers, sentinels, and other types of monitors that assess the health of other elements or of the system as a whole.

Traditionally, many of these services have been part of the system infrastructure; in a multiagent, autonomic world, moving them out of the infrastructure and representing them as autonomic elements themselves will be more natural and flexible.

Goal specification

While autonomic systems will assume much of the burden of system operation and integration, it will still be up to humans to provide those systems with policies—the goals and constraints that govern their actions. The enormous leverage of autonomic systems will greatly reduce human errors, but it will also greatly magnify the consequences of any error humans do make in specifying goals.

The indirect effect of policies on system configuration and behavior exacerbates the problem because tracing and correcting policy errors will be very difficult. It is thus critical to ensure that the specified goals represent what is really desired. Two engineering challenges stem from this mandate: Ensure that goals are specified correctly in the first place, and ensure that systems behave reasonably even when they are not.

In many cases, the set of goals to be specified will be complex, multidimensional, and conflicting. Even a goal as superficially simple as "maximize utility" will require a human to express a complicated multiattribute utility function. A key to reducing error will be to simplify and clarify the means by which humans express their goals to computers. Psychologists and computer scientists will need to work together to strike the right balance between overwhelming humans with too many questions or too much information and underempowering them with too few options or too little information.

The second challenge—ensuring reasonable system behavior in the face of erroneous input—is another facet of robustness: Autonomic systems will need to protect themselves from input goals that are inconsistent, implausible, dangerous, or unrealizable with the resources at hand. Autonomic systems will subject such inputs to extra validation, and when self-protective measures fail, they will rely on deep-seated notions of what constitutes acceptable behavior to detect and correct problems. In some cases, such as resource overload, they will inform human operators about the nature of the problem and offer alternative solutions.

SCIENTIFIC CHALLENGES

The success of autonomic computing will hinge on the extent to which theorists can identify universal principles that span the multiple levels at which autonomic systems can exist—from systems to enterprises to economies.

Behavioral abstractions and models

Defining appropriate abstractions and models for understanding, controlling, and designing emergent behavior in autonomic systems is a challenge at the heart of autonomic computing. We need fundamental mathematical work aimed at understanding how the properties of self-configuration, self-optimization, self-maintenance, and robustness arise from or depend on the behaviors, goals, and adaptivity of individual autonomic elements; the pattern and type of interactions among them; and the external influences or demands on the system.

Understanding the mapping from local behavior to global behavior is a necessary but insufficient condition for controlling and designing autonomic systems. We must also discover how to exploit the inverse relationship: How can we derive a set of behavioral and interaction rules that, if embedded in individual autonomic elements, will induce a desired global behavior? The nonlinearity of emergent behavior makes such an inversion highly nontrivial.

One plausible approach couples advanced search and optimization techniques with parameterized models of the local-to-global relationship and the likely set of environmental influences to which the system will be subjected. Melanie Mitchell and colleagues[9] at the Santa Fe Institute have pioneered this approach, using genetic algorithms to evolve the local transformation rules of simple cellular automata to achieve desired global behaviors. At NASA, David Wolpert and colleagues[10] have studied algorithms that, given a high-level global objective, derive individual goals for individual agents. When each agent selfishly follows its goals, the desired global behavior results.

These methods are just a start. We have yet to understand fundamental limits on what classes of

global behavior can be achieved, nor do we have practical methods for designing emergent system behavior. Moreover, although these methods establish the rules of a system at design time, autonomic systems must deal with shifting conditions that can be known only at runtime. Control theoretic approaches may prove useful in this capacity; some autonomic managers may use control systems to govern the behavior of their associated managed elements.

The greatest value may be in extending distributed or hierarchical control theories, which consider interactions among independently or hierarchically controlled elements, rather than focusing on an individual controlled element. Newer paradigms for control may be needed when there is no clear separation of scope or time scale.

Robustness theory

A related challenge is to develop a theory of robustness for autonomic systems, including definitions and analyses of robustness, diversity, redundancy, and optimality and their relationship to one another. The Santa Fe Institute recently began a multidisciplinary study on this topic (http://discuss.santafe.edu/robustness).

Learning and optimization theory

Machine learning by a single agent in relatively static environments is well studied, and it is well supported by strong theoretical results. However, in more sophisticated autonomic systems, individual elements will be agents that continually adapt to their environment—an environment that consists largely of other agents. Thus, even with stable external conditions, agents are adapting to one another, which violates the traditional assumptions on which single-agent learning theories are based.

There are no guarantees of convergence. In fact, interesting forms of instability have been observed in such cases.[11] Learning in multiagent systems is a challenging but relatively unexplored problem, with virtually no major theorems and only a handful of empirical results.

Just as learning becomes a more challenging problem in multiagent systems, so does optimization. The root cause is the same—whether it is because they are learning or because they are optimizing, agents are changing their behavior, making it necessary for other agents to change their behavior, potentially leading to instabilities. Optimization in such an environment must deal with dynamics created by a collective mode of oscillation rather than a drifting environmental signal. Optimization techniques that assume a stationary environment have been observed to fail pathologically in multiagent systems,[12] therefore they must either be revamped or replaced with new methods.

Optimization techniques that assume a stationary environment must either be revamped or replaced with new methods.

Negotiation theory

A solid theoretical foundation for negotiation must take into account two perspectives. From the perspective of individual elements, we must develop and analyze algorithms and negotiation protocols and determine what bidding or negotiation algorithms are most effective.

From the perspective of the system as a whole, we must establish how overall system behavior depends on the mixture of negotiation algorithms that various autonomic elements use and establish the conditions under which multilateral—as opposed to bilateral—negotiations among elements are necessary or desirable.

Automated statistical modeling

Statistical models of large networked systems will let autonomic elements or systems detect or predict overall performance problems from a stream of sensor data from individual devices. At long time scales—during which the configuration of the system changes—we seek methods that automate the aggregation of statistical variables to reduce the dimensionality of the problem to a size that is amenable to adaptive learning and optimization techniques that operate on shorter time scales.

Is it possible to meet the grand challenge of autonomic computing without magic and without fully solving the AI problem? We believe it is, but it will take time and patience. Long before we solve many of the more challenging problems, less automated realizations of autonomic systems will be extremely valuable, and their value will increase substantially as autonomic computing technology improves and earns greater trust and acceptance.

A vision this large requires that we pool expertise in many areas of computer science as well as in disciplines that lie far beyond computing's traditional boundaries. We must look to scientists studying nonlinear dynamics and complexity for new theories of emergent phenomena and robustness. We must look to economists and e-commerce researchers for ideas and technologies about negotiation and supply webs. We must look to psychologists and human factors researchers for new goal-definition and visualization paradigms and

for ways to help humans build trust in autonomic systems. We must look to the legal profession, since many of the same issues that arise in the context of e-commerce will be important in autonomic systems that span organizational or national boundaries.

Bridging the language and cultural divides among the many disciplines needed for this endeavor and harnessing the diversity to yield successful and perhaps universal approaches to autonomic computing will perhaps be the greatest challenge. It will be interesting to see what new cross-disciplines develop as we begin to work together to solve these fundamental problems. ■

Acknowledgments

We are indebted to the many people who influenced this article with their ideas and thoughtful criticisms. Special thanks go to David Chambliss for contributing valuable thoughts on human-computer interface issues. We also thank Bill Arnold, David Bantz, Rob Barrett, Peter Capek, Alan Ganek, German Goldszmidt, James Hanson, Joseph Hellerstein, James Kozloski, Herb Lee, Charles Peck, Ed Snible, and Ian Whalley for their helpful comments, and the members of an IBM Academy of Technology team for their extensive written and verbal contributions: Lisa Spainhower and Kazuo Iwano (co-leaders), William H. Tetzlaff (Technology Council contact), Robert Abrams, Sam Adams, Steve Burbeck, Bill Chung, Denise Y. Dyko, Stuart Feldman, Lorraine Herger, Mark Johnson, James Kaufman, David Kra, Ed Lassettre, Andreas Maier, Timothy Marchini, Norm Pass, Colin Powell, Stephen A. Smithers, Daniel Sturman, Mark N. Wegman, Steve R. White, and Daniel Yellin.

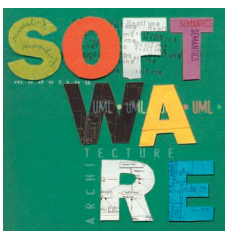
References

1. IBM, "Autonomic Computing: IBM's Perspective on the State of Information Technology"; http://www-1.ibm.com/industries/government/doc/content/resource/thought/278606109.html.
2. H. Kreger, "Web Services Conceptual Architecture," v. 1.0. 2001; http://www-4.ibm.com/software/solutions/webservices/pdf/WSCA.pdf.
3. I. Foster et al., "The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration," Feb. 2002; http://www.globus.org/research/papers/ogsa.pdf.
4. N.R. Jennings, "On Agent-Based Software Engineering," *Artificial Intelligence*, vol. 177, no. 2, 2000, pp. 277-296.
5. D. Patterson et al., *Recovery-Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies*, tech. report CSD-02-1175, Computer Science Dept., Univ. of Calif., Berkeley, Calif., Mar. 2002.
6. Ariba, IBM, and Microsoft, "UDDI Technical White Paper," 2000; http://www.uddi.org/whitepapers.html.
7. T. Berners-Lee, J. Hendler, and O. Lassila, "The Semantic Web," *Scientific American*, May 2001, pp. 28-37.
8. H. Wong and K. Sycara, "A Taxonomy of Middle Agents for the Internet," *Proc. 4th Int'l Conf. Multiagent Systems*, IEEE CS Press, 2000, pp. 465-466.
9. R. Das et al., "Evolving Globally Synchronized Cellular Automata," *Proc. 6th Int'l Conf. Genetic Algorithms*, L. Eshelman, ed., Morgan Kaufmann, 1995, pp. 336-343.
10. D. Wolpert, K. Wheeler, and K. Tumer, *Collective Intelligence for Control of Distributed Dynamical Systems*, tech. report NASA-ARC-IC-99-44, NASA, Ames, Iowa, 1999.
11. J.O. Kephart and G.J. Tesauro, "Pseudo-Convergent Q-Learning by Competitive Pricebots," *Proc. 17th Int'l Conf. Machine Learning*, Morgan Kaufmann, 2000, pp. 463-470.
12. J.O. Kephart et al., "Pricing Information Bundles in a Dynamic Environment," *Proc. 3rd ACM Conf. Electronic Commerce*, 2001, ACM Press, pp. 180-190.

Jeffrey O. Kephart *manages the Agents and Emergent Phenomena group at the IBM Thomas J. Watson Research Center. His research focuses on the application of analogies from biology and economics to massively distributed computing systems, particularly in the domains of autonomic computing, e-commerce, and antivirus technology. Kephart received a BS from Princeton University and a PhD from Stanford University, both in electrical engineering. Contact him at kephart@us.ibm.com.*

David M. Chess *is a research staff member at the IBM Thomas J. Watson Research Center, working in autonomic computing and computer security. He received a BA in philosophy from Princeton University and an MS in computer science from Pace University. Contact him at chess@us.ibm.com.*

Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure



The Rainbow framework uses software architectures and a reusable infrastructure to support self-adaptation of software systems. The use of external adaptation mechanisms allows the explicit specification of adaptation strategies for multiple system concerns.

David Garlan
Shang-Wen Cheng
An-Cheng Huang
Bradley Schmerl
Peter Steenkiste
Carnegie Mellon University

Software-based systems today increasingly operate in changing environments with variable user needs, resulting in the continued increase in administrative overhead for managing these systems. To reduce these costs, systems are increasingly expected to dynamically self-adapt to accommodate resource variability, changing user needs, and system faults. Mechanisms that support self-adaptation currently exist in the form of programming language features such as exceptions and in algorithms such as fault-tolerant protocols. But these mechanisms are often highly specific to the application and tightly bound to the code. As a result, self-adaptation in today's systems is costly to build, difficult to modify, and usually provides only localized treatment of system faults.

In contrast to these internal mechanisms, recent work uses external models and mechanisms in a closed-loop control fashion to achieve various goals by monitoring and adapting system behavior at runtime.[1,2] As illustrated in Figure 1, control of system adaptation becomes the responsibility of components outside the system that is being adapted.

In principle, external control mechanisms provide a more effective engineering solution than internal mechanisms for self-adaptation because they localize the concerns of problem detection and resolution in separable modules that can be analyzed, modified, extended, and reused across different systems. Additionally, developers can use this approach to add self-adaptation to legacy systems for which the source code may not be available.

This external approach requires using an appropriate model to reason about the system's dynamic behavior. Several researchers have proposed using architectural models,[3] which represent the system as a gross composition of components, their interconnections, and their properties of interest.[4] Such an *architecture-based self-adaptation* approach offers many benefits. Most significantly, an abstract architectural model can provide a global perspective of the system and expose important system-level properties and integrity constraints.

While attractive in principle, architecture-based self-adaptation raises a number of research and engineering challenges. First, the ability to handle a wide variety of systems must be addressed. Since different systems have radically different architectural styles, properties of interest, and mechanisms supporting dynamic modification, it is critical that the architectural control model and modification strategies be tailored to the specific system. Second, the need to reduce costs in adding external control to a system must be addressed. Creating the monitoring, modeling, and problem-detection mecha-

Published by the IEEE Computer Society

nisms from scratch for each new system would render the approach prohibitively expensive.

Our Rainbow framework attempts to address both problems. By adopting an architecture-based approach, it provides reusable infrastructure together with mechanisms for specializing that infrastructure to the needs of specific systems. These specialization mechanisms let the developer of self-adaptation capabilities choose what aspects of the system to model and monitor, what conditions should trigger adaptation, and how to adapt the system.

THE RAINBOW FRAMEWORK

Figure 2 shows the Rainbow framework's control loop for self-adaptation. Rainbow uses an abstract architectural model to monitor an executing system's runtime properties, evaluates the model for constraint violation, and—if a problem occurs—performs global- and module-level adaptations on the running system.

Software architectures

Rainbow adopts a standard view of software architecture that is typically used today at design time to characterize a system to be built. Specifically, an architecture is represented as a graph of interacting computational elements.[4] Nodes in the graph, called *components*, represent the system's principal computational elements and data stores, including clients, servers, databases, and user interfaces. Arcs, called *connectors*, represent the pathways for interaction between the components. Additionally, architectural elements may be annotated with various properties, such as expected throughputs, latencies, and protocols of interaction. Components themselves may represent complex systems, which are represented hierarchically as subarchitectures.

However, unlike traditional uses of software architecture as strictly a design-time artifact, Rainbow includes a system's architectural model in its runtime system. In particular, developers of self-adaptation capabilities use a system's software architectural model to monitor and reason about the system. Using a system's architecture as a control model for self-adaptation holds promise in several areas. As an abstract model, an architecture can provide a global perspective of the system and expose important system-level behaviors and properties. As a locus of high-level system design decisions, an architectural model can make a system's topological and behavioral constraints explicit, establishing an envelope of allowed changes and helping to ensure the validity of a change.

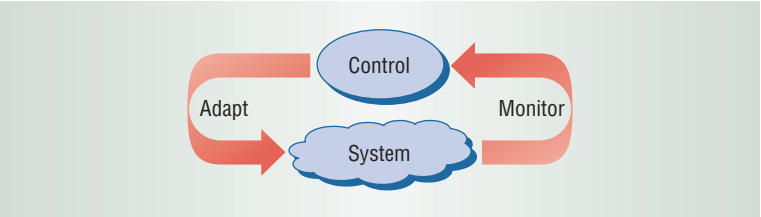


Figure 1. External control of self-adaptation uses external models to monitor and modify a system dynamically.

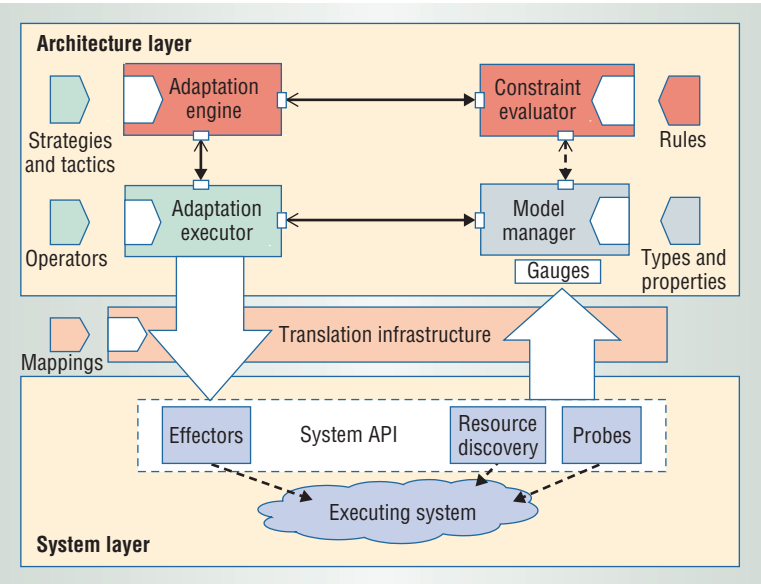


Figure 2. Rainbow framework. The framework uses an abstract model to monitor an executing system's runtime properties, evaluates the model for constraint violation, and—if a problem occurs—performs adaptations on the running system.

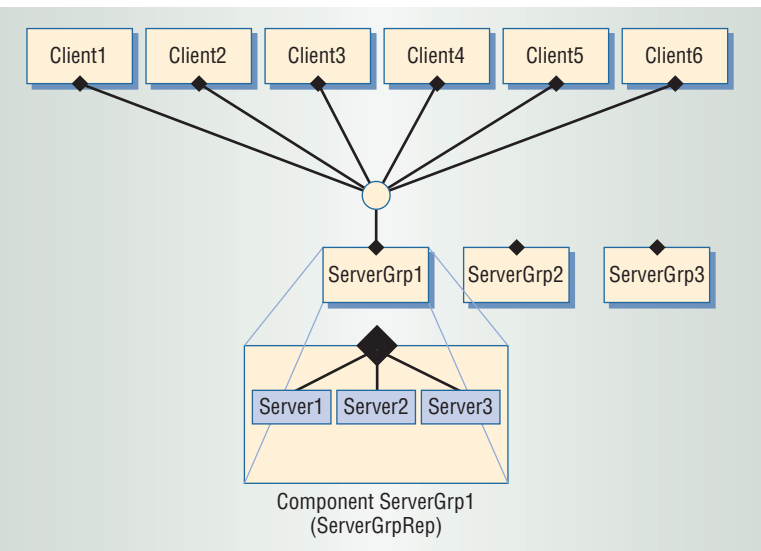


Figure 3. Client-server system software architecture. This model represents the architecture as a hierarchical graph of interacting components.

Figure 3 shows one example of an architecture in which the components represent Web clients and server clusters. Each server cluster has a subarchi-

To capture system commonalities, Rainbow adopts the notion of an *architectural style*, which describes a family of systems related by shared structural and semantic properties.

tecture consisting of one or more server components. This architectural model provides a global perspective on the system by revealing all the components and how they connect. The model also contains important properties such as each server's load, each connection's bandwidth, and the response time experienced by each client.

Further, the model maintains explicit constraints on the architecture that, for example, require each client to connect to exactly one server cluster. The constraints establish an envelope of allowed changes such as ensuring that no future changes to the system leave a client dangling without a connection. A system change is valid only if the system satisfies the constraints after the change.

Reusable Rainbow units

To fulfill Rainbow's objectives, its various components must be reusable from system to system. To identify what parts of the framework are reusable, and under what circumstances, we divide the framework into an adaptation infrastructure and the system-specific adaptation knowledge. The adaptation infrastructure, divided into system, architecture, and translation layers, provides common functionalities across self-adapting systems and is therefore reusable across all systems, while the adaptation knowledge itself is typically system-specific (Figure 2).

System-layer infrastructure. At this layer, we have defined the system access interface and built an infrastructure that implements it. A system measurement mechanism, realized as *probes*, observes and measures various system states. This low-level system information can be published by or queried from the probes. Additionally, a *resource discovery* mechanism can be queried for new resources based on resource type and other criteria. Finally, an *effector* mechanism carries out the actual system modification.

Architecture-layer infrastructure. At this layer, *gauges* aggregate information from the probes and update the appropriate properties in the architectural model. A *model manager* handles and provides access to the system's architectural model. A *constraint evaluator* checks the model periodically and triggers adaptation if a constraint violation occurs. An *adaptation engine* then determines the course of action and carries out the necessary adaptation.

Translation infrastructure. This infrastructure helps mediate the mapping of information across the abstraction gap from the system to the model and vice versa. A *translation repository* within the infrastructure maintains various mappings that the *translator components* share, for example, to translate an architectural-level element identifier into an IP address or an architectural-level change operator into system-level operations.

System-specific adaptation knowledge. Adding self-adaptation to a system using the functionalities that the adaptation infrastructure provides requires using the system-specific adaptation knowledge to tailor that infrastructure. This knowledge includes the target system's operational model, which defines parameters such as component types and properties, behavioral constraints, and adaptation strategies.

Architectural style

While reusable infrastructure helps reduce the costs of adding self-adaptation to systems, it is also possible to leverage commonalities in system architecture to encapsulate adaptation knowledge for various system classes.

To capture system commonalities, Rainbow adopts the notion of an *architectural style*. Traditionally, the software engineering community has used architectural styles to help encode and express system-specific knowledge.[5] An architectural style characterizes a family of systems related by shared structural and semantic properties. The style is typically defined by four sets of entities:

- *Component and connector types* provide a vocabulary of elements, including components such as Database, Client, Server, and Filter; connectors such as SQL, HTTP, RPC, and Pipe; and component and connector interfaces.
- *Constraints* determine the permitted composition of the elements instantiated from the types. For example, constraints might prohibit cycles in a particular pipe-filter style, or define a compositional pattern such as the starfish arrangement of a blackboard system or a compiler's pipelined decomposition.
- *Properties* are attributes of the component and connector types, and provide analytic, behavioral, or semantic information. For example, load and service time properties might be characteristic of servers in a performance-specific client-server style, while transfer-rate might be a property in a pipe-filter style.
- *Analyses* can be performed on systems built in an appropriate architectural style. Examples include performance analysis using queuing theory in a client-server system, and schedulability analysis for a real-time-oriented style.

These four entity sets primarily capture a system's static attributes. Rainbow extends this notion of architectural style to support runtime adaptation by also capturing the system's dynamic attributes, both in terms of the primitive operations that can be performed on the system to change it dynamically, and how the system can combine those operations to achieve some effect. Specifically, it augments the notion of architectural style with *adaptation operators* and *strategies*, which together determine the system's *adaptation style*.

- *Adaptation operators* determine a set of style-specific actions that the control infrastructure can perform on a system's elements to alter its configuration. For example, a service coalition style might define the operators AddService or RemoveService to add or remove services from a system configuration in this style.
- *Adaptation strategies* specify the adaptations that can be applied to move a system away from an undesirable condition. For example, a service-coalition system might have a system-wide cost constraint. Upon violating it, an adaptation strategy might progressively replace the most costly service with lower-grade services until the overall cost falls within acceptable bounds. Strategies are defined using—and therefore constrained by—operators and properties.

Although strategies use operators and properties to adapt systems of a particular style, they are designed for particular *system concerns*. A system concern outlines a related set of system requirements—such as performance, cost, or reliability—and determines the set of system properties on which self-adaptation should focus, and hence the set of strategies. The system concerns form a subset of the properties in a system's style. For example, a client-server system may have a style that includes load, bandwidth, and cost properties, while a particular performance concern might focus only on the system's load and bandwidth properties.

Adaptation style and system concerns together comprise two important dimensions of variability from system to system. How much of Rainbow's system-specific adaptation knowledge can be reused will depend on how similar two systems are in terms of their styles and system concerns. More specifically, types, properties, adaptation strategies, and operators may be reusable if the two systems have matching styles and concerns. Two case studies help illustrate this concept.

ADAPTATION CASE STUDIES

The following studies examine two systems with different adaptation styles that share the same system concern. This allows reusing the Rainbow framework across both prototype systems.

Rainbow supports runtime adaptation by also capturing the system's dynamic attributes.

Web-based client-server system

The first case study system consists of a set of Web clients, each of which makes stateless requests of contents from one of several Web server groups, as Figure 3 shows. The client and server components are implemented in Java and provide remote method invocation (RMI) interfaces for the effectors to use in performing adaptation operations. Clients connected to a server group send requests to the group's shared request queue, and servers that belong to the group grab requests from the queue.

The system concern focuses primarily on performance—specifically, the response time the clients experience. A queuing theory analysis of the system identifies that the server load and available bandwidth are two properties that affect the response time. Based on this system concern and analysis, the developer defines a client-server style for the system. The major parts of the style include

- ClientT, ServerT, ServerGroupT, and LinkT *types*;
- ClientT.responseTime, ServerT.load, ServerGroupT.load, and LinkT.bandwidth *properties*; and
- ServerGroupT.addServer() and ClientT.move (ServerGroupT, toGroup) *operators*.

The ServerGroupT.addServer() operator finds and adds an available ServerT to a ServerGroupT to increase the capacity. The ClientT.move (ServerGroupT, toGroup) operator disconnects ClientT from its current ServerGroupT, then connects ClientT to the toGroup ServerGroupT.

Associated with each client is an invariant that checks to see if its perceived response time is less than a predefined maximum response time. If the invariant fails, an adaptation strategy is invoked. An example invariant and adaptation strategy is

```
invariant (self.responseTime <
           maxResponseTime)
  !→ responseTimeStrategy(self);

strategy responseTimeStrategy
```

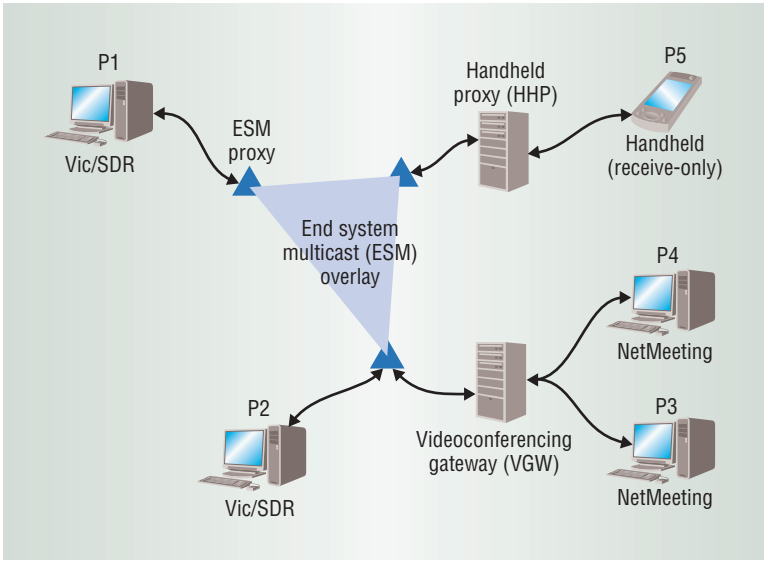


Figure 4. Videoconferencing adaptation case study. Two participants use Vic/SDR IP multicast videoconferencing tools; two others use NetMeeting, which adopts the H.323 protocol and unicast; while the final participant uses a handheld device running a slightly modified version of Vic.

```
            (ClientT C) {
let G = findConnectedServerGroup
            (C);
if (query("load", G) >
            maxServerLoad) {
      G.addServer();
      return true;
}
let conn = findConnector(C, G);
if (query("bandwidth", conn) <
            minBandwidth) {
      let G = findBestServerGroup
            (C);
      C.move(G);
      return true;
  }
return false;
}
```

In this specification, the invariant defines a predicate that determines whether a client's perceived response time (self.responseTime) is below a threshold (maxResponseTime). If this invariant is violated (indicated by "!→"), the adaptation engine executes the strategy responseTimeStrategy.

This strategy first checks to see if the current server group's load exceeds a predefined threshold. If so, the engine adds a server to the group to decrease the load and thus decrease response time. If, however, the available bandwidth between the client and the current server group drops too low, the engine moves the client to another group, resulting in higher available bandwidth and lower response time.

Videoconferencing system

Figure 4 shows the second system example, a videoconferencing session with five participating users. Two of the participants use the Vic/SDR videoconferencing tools, which use the Session Initiation Protocol (SIP) and IP multicast. Two other participants use NetMeeting, which uses the H.323 protocol and unicast. The final participant uses a handheld device, which runs a slightly modified version of Vic.

Since the handheld device cannot perform protocol negotiation, a handheld proxy (HHP) joins the conferencing session on behalf of the handheld user. A videoconferencing gateway (VGW) that supports both H.323 and SIP translates the protocols for NetMeeting and Vic users. Finally, to allow efficient communication among all participants across wide-area networks, the system uses Narada, an end-system multicast overlay consisting of three proxies, to provide the multicast functionality.

The system concerns here involve both performance and cost. For example, the system seeks to maintain sufficient available bandwidth between the handheld user and the handheld proxy, while keeping the cost of providing the videoconferencing service low. For example, if only one NetMeeting user remains online, the system should switch to a low-cost gateway. The developers define a videoconferencing style for the system based on these concerns. The major parts of the style include the following:

- VicT, NetMeetingT, HandheldT, GatewayT, HandheldProxyT, ESMProxyT, and ConnectionT *component and connector types*;
- GatewayT.cost, GatewayT.load, and ConnectionT.bandwidth *properties*; and
- HandheldT.move(HandheldProxyT, toHHP) and NetMeetingT.move(GatewayT, toVGW) *operators*.

In this case, the HandheldT.move(HandheldProxyT, toHHP) operator switches the handheld user to a new handheld proxy, while the NetMeetingT.move(GatewayT, toVGW) operator switches the NetMeeting user to a new video gateway.

Two sample adaptation strategies specify the desired adaptive behavior:

```
invariant (bandwidthToHHP (self)
          > minHHBandwidth)
  !→ HHBandwidthStrategy(self);

invariant (self.cost /
          numberOfNMusers <
          maxVGWUnitCost)
  !→ VGWCostStrategy(self);

strategy HHBandwidthStrategy
           (HandheldT HH) {
  let HHP1 = findBestHHP(HH);
  HH.move(HHP1);
  return true;
}

strategy VGWCostStrategy
           (GatewayT VGW) {
  let VGW1 = the gateway with the
           lowest cost that
           can handle the
           current load;
  if ((query("cost", VGW1) /
           numberofNMusers)
      < maxVGWUnitCost) {
    foreach NetMeeting user U of
           VGW {
      U.move(VGW1);
    }
    return true;
  }
  return false;
}
```

The first invariant is associated with components of type HandheldT, while the second invariant is associated with components of type GatewayT.

At runtime, when either invariant is violated, the adaptation engine executes the corresponding adaptation strategy. For example, when the available bandwidth between the handheld user and the handheld proxy drops too low, the engine moves the handheld user to a better handheld proxy.

When the unit cost of the gateway VGW becomes too high—for example, when a NetMeeting user leaves the session—the engine switches the NetMeeting users connected to VGW to the lowest-cost gateway that can handle the load.

A conflict between concerns of performance and cost is possible, such as when an adaptation pushes the system's total service cost above a maximum threshold. Although not addressed here, a composite utility function can help resolve such conflicts.

Reuse analysis

Several reuse issues can be better understood by examining how the two case study systems reuse the adaptation infrastructure's three layers and the various parts of style, which represent the system-specific adaptation knowledge.

System layer infrastructure. This layer consists of three elements: effectors, probes, and resource discovery. *Effectors* are component-specific: An effector can perform adaptation operations only on one type of system component. For example, a videoconferencing gateway effector can activate and shut down a gateway. Rainbow's system-layer infrastructure provides a reusable interface for accessing the effectors. For example, the adaptation engine/executor can issue an Activate operation by invoking the corresponding function that the system application programming interface (API) provides. The system- layer infrastructure then dispatches the operation to the appropriate effector based on the target component's type. Because the two case study systems do not share any components, their effectors cannot be reused.

The system-layer infrastructure provides *probes* that measure the response time, load, and bandwidth of various system components. Among these, the load and bandwidth probes are reused across the two systems because these two properties are of interest in both systems. Probes support monitoring and querying of information that is used at higher levels of the infrastructure to update model properties. In addition, the adaptation strategies often need to query some additional information, such as the server group load and new gateway's cost.

The adaptation engine needs a *resource discovery* mechanism to find available components to replace existing ones as directed by the adaptation strategy. For example, the first strategy for the videoconferencing system requires the adaptation engine to find the HHP component with the most available bandwidth for the handheld user. The second strategy finds a VGW component that supports both Vic users and NetMeeting users and has the lowest cost. Rainbow's system-layer infrastructure supports resource discovery based on component type and other desired component attributes.

Architecture-layer infrastructure. At the framework's architecture layer, the adaptation style spec-

At the framework's architecture layer, the adaptation style specifies the model instance's types, properties, and rules.

Table 1. Framework system-specific adaptation knowledge reuse summary.

Architectural style	System concerns	Reuse achieved
Different	Different	Adaptation infrastructure
Different	Same	Adaptation infrastructure, Properties, Mappings
Same	Different	Adaptation infrastructure, Types, Rules, Mappings, Adaptation operators
Same	Same	Adaptation infrastructure, Types, Rules, Properties, Mappings, Adaptation operators

ifies the types, properties, and rules of the model instance that the model manager handles. The functionalities of the gauges, model manager, constraint evaluator, and adaptation engine remain the same. Gauges for the properties of response time, load, and bandwidth aggregate information from the corresponding probes and update the appropriate properties in the model.

Translation infrastructure. This layer bridges the abstraction gap between the model and the system. For example, when the adaptation engine performs resource discovery to find a VGW, the translation layer must map the architectural type GatewayT to the system-level component type sysGateway that the system-layer resource discovery mechanism uses. Likewise, when the adaptation engine applies the connect operator to two elements in the architectural model, such as a NetMeeting user and a gateway, the translation layer must map them to actual machines in the system. These mappings are stored in the translation repository, and the translators perform the actual translation.

System-specific adaptation knowledge. The two case study systems differ in adaptation styles but share a system concern. This concern manifests itself in the properties of each style, so sharing the same concern means that the system can reuse the knowledge about the shared properties of load and bandwidth. Part of the knowledge is the translation mappings between the system properties and architectural properties that the style defines. For properties that the gauges need to aggregate, such as average latency, the aggregation knowledge can also be reused.

Drawing on these two case studies, Table 1 generalizes from our experience in determining what system-specific adaptation knowledge in the framework can be reused depending on the two dimensions of adaptation style and system concerns.

Implementation and evaluation

Moving beyond the two case studies, we have implemented a prototype of the Rainbow self-adaptation framework. At the system level, we use the global network positioning (GNP) approach to estimate network latency, the Remos tool[6] to measure bandwidth, and the network-sensitive service discovery (NSSD) mechanism[7] to discover resources. We implemented probes that obtain information from GNP and the Remos tool.

The architecture-layer entities are implemented in Java, based on the Acme architectural design toolset.[8] Performance-property gauges were implemented to read values from the probes and update the model manager's model via two event broadcast buses. For the translation infrastructure, the translation repository provides a Java RMI interface for the translators to use for storing and retrieving the necessary translation mappings.

Some translators are stand-alone entities, while others are integrated modules of system-layer or architecture-layer entities. Communications within the framework use XML messages over Java RMI.

Turning to the issue of reuse, although code size is not the only reuse measure, we can use the Rainbow prototype's code size to approximate the degree of reuse for the two systems. The Rainbow prototype—including the adaptation mechanism; model manager; gauge, probes, and their infrastructure; and the translation and system-layer infrastructure—requires 102 kilolines of code (KLoCs), with a breakdown of 84, 11, and 4 KLoCs for the architecture, system, and translation layers, respectively. Of these, the nonreused code and data for adaptation and translation mappings occupy about 1.8 KLoCs. In addition, the project reused 73 KLoCs of tool and utility code.

The Rainbow framework's self-adaptation effectiveness and performance are two other important aspects requiring evaluation. We can use the client-server system to demonstrate the effectiveness of the framework's self-adaptation. To demonstrate this, we conducted an experiment on a dedicated testbed consisting of five routers and 11 machines communicating over 10-megabits-per-second lines. This experiment conducted repairs, while the network was overloaded, on a client-server system that required a client latency of less than two seconds.

The results show that for this application and the specific loads used in the experiment, self-repair significantly improved system performance. Figure 5 shows sample results for system perfor-

mance with and without adaptation. Figure 5a shows that, without adaptation, once the latency experienced by each client rises above 2 seconds, it never again falls below this threshold. On the other hand, Figure 5b shows that if Rainbow issues the repairs, the client latencies return to optimal levels.

Perhaps not unexpectedly, our experiment also revealed that external repair has an associated latency. In the client-server example, it took several seconds for the system to notice a performance problem and several more seconds to fix it. For the videoconferencing application, elapsed time for adaptation at the architecture, translation, and system layers were 230, 300, and 1,600 ms, respectively, for one scenario, and 330, 900, and 1,500 ms, respectively, for another. Although we can imagine speeding up the round-trip repair time, these results indicate that the software architecture-based approach best suits repairs that operate on a systemwide scale and fix longer-term system behavior trends.

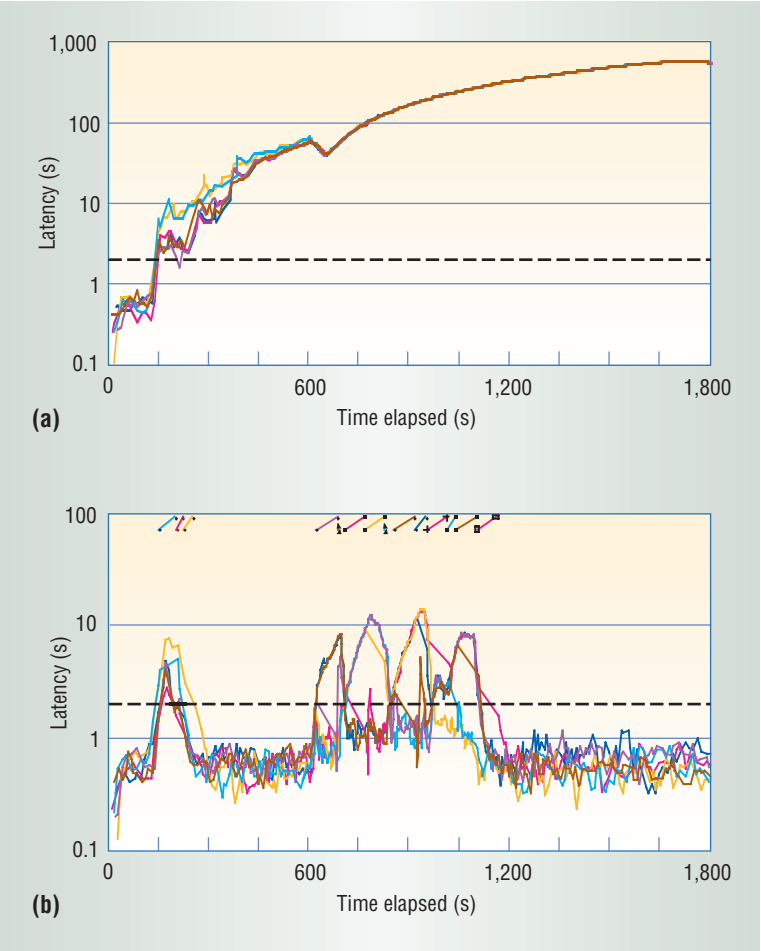


Figure 5. System performance with and without adaptation. The dashed lines indicate the desired latency behavior. (a) Without adaptation, if each client's latency rises above 2 seconds, it never again falls below that threshold. (b) Once repaired, client latencies soon return to optimal levels.

Although our evaluation indicates that it is possible to achieve architecture-based self-adaptation at low cost using reusable infrastructure, it is worth noting that the Rainbow framework rests on some important assumptions.

One assumption is that any target system will provide system access hooks for monitoring and adaptation. This assumption seems reasonable, at least for measurement, because an increasing number of measurement tools and infrastructure provide measures of or information about common component properties, including network bandwidth and latency. Several protocols can discover new services and resources for a system. Likewise, emerging effector technologies support dynamic changes to running system components. Also, developers can use wrappers to add hooks for making changes in legacy systems.

Another assumption lies in the adaptation infrastructure, where we assume probes will provide information about system properties, along with gauges to aggregate the information and update properties in the model manager. This assumption is based on a model that has well-defined gauge and probe APIs. We anticipate that external developers will specialize in developing gauges and probes for various purposes. Further, the measurement tools we have described could also implement the probe API or be wrapped to serve as probes, which would let developers of self-adapting systems plug in any gauge and probe to suit their needs.

Finally, the work we have described is inherently centralized, with monitoring and adaptation performed within a single Rainbow instance. Making this assumption has let us focus on core issues of self-adaptation—specifically monitoring, detection, resolution, and adaptation. At the same time, there may be concerns regarding scalability and single-point failure. The Rainbow framework can, however, be applied in a distributed setting. For example, we could apply Rainbow instances to adapt multiple subsystems of a distributed system, and then coordinate those instances toward an overall adaptation goal. The coordination and other distributed computing issues present a challenge for future research. ■

Acknowledgments

This research was supported by DARPA under grants N66001-99-2-8918 and F30602-00-2-0616, by the US Army Research Office (ARO) under grant number DAAD19-01-1-0485, and the NASA High Dependability Computing Program under cooperative agreement NCC-2-1298. The views and conclusions described here are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of DARPA, the ARO, NASA, the US government, or any other entity.

References

1. D. Garlan, J. Kramer, and A. Wolf, eds., *Proc. 1st ACM SIGSOFT Workshop on Self-Healing Systems* (WOSS 02), ACM Press, 2002.
2. A.G. Ganak and T.A. Corbi, "The Dawning of the Autonomic Computing Era, *IBM Systems J.*, vol. 42, no. 1, 2003, pp. 5-18.
3. P. Oriezy et al., "An Architecture-Based Approach to Self-Adaptive Software," *IEEE Intelligent Systems*, vol. 14, no. 3, 1999, pp. 54-62.
4. P. Clements et al., *Documenting Software Architecture: Views and Beyond*, Addison-Wesley, 2003.
5. M. Shaw and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, 1996.
6. T. Gross et al., "Design, Implementation, and Evaluation of the Remos Network Monitoring System," *J. Grid Computing*, vol. 1, no. 1, 2003, pp. 75-93.
7. A.-C. Huang and P. Steenkiste, "Network-Sensitive Service Discovery," *J. Grid Computing*, vol. 1, no. 1, 2003; www.cs.cmu.edu/~pach.
8. D. Garlan, R.T. Monroe, and D. Wile, "Acme: Architectural Descriptions of Component-Based Systems," *Foundations of Component-Based Systems*, G.T. Leavens and M. Sitaraman, eds., Cambridge Univ. Press, 2000, pp. 47-68.

David Garlan *is a professor of computer science at Carnegie Mellon University. His research interests include software architectures, formal methods, self-healing systems, and task-based computing. He received a PhD in computer science from Carnegie Mellon University. Contact him at garlan@cs.cmu.edu.*

Shang-Wen Cheng *is a doctoral candidate at Carnegie Mellon University. His research interests include dynamic system adaptation, software architectures, and software designs for security. He received a BS in computer information sciences from Florida State University. Contact him at chengs@cmu.edu.*

An-Cheng Huang *is a doctoral candidate at Carnegie Mellon University. His research interests include distributed systems, networking, and grid computing. He received a BS in computer science and information engineering from National Taiwan University. Contact him at pach@cs.cmu.edu.*

Bradley Schmerl *is a systems scientist at Carnegie Mellon University. His research interests include dynamic adaptation, software architectures, and software engineering environments. He received a PhD in computer science from Flinders University in South Australia. Contact him at schmerl@cs.cmu.edu.*

Peter Steenkiste *is a professor of computer science and electrical and computer engineering at Carnegie Mellon University. His research interests include networking and distributed systems. He received a PhD in electrical engineering from Stanford University. Contact him at prs@cs.cmu.edu.*

Self-Managed Systems: an Architectural Challenge

Jeff Kramer and Jeff Magee

Jeff Kramer is Dean of the Faculty of Engineering at Imperial College London, and was Head of the Department of Computing from 1999-2004. His research interests include rigorous techniques for requirements engineering; software specification, design and analysis; and software architectures, particularly as applied to distributed and adaptive software systems. Jeff is the Editor-in-Chief of the IEEE Transactions on Software Engineering, and the co-recipient of the 2005 ACM SIGSOFT Outstanding Research Award for his work in Distributed Software Engineering. He is co-author of a recent book on Concurrency, co-author of a previous book on Distributed Systems and Computer Networks, and the author of over 200 journal and conference publications. He is a Chartered Engineer, Fellow of the IET, Fellow of the BCS and Fellow of the ACM.

Jeff Magee is Head of the Department of Computing at Imperial College London. His research is primarily concerned with the software engineering of distributed systems, including requirements, design methods, analysis techniques, operating systems, languages and program support environments for these systems. He is co-author of a recent book on concurrent programming entitled "Concurrency - State models and Java programs" and the author of too many journal and conference publications. He was co-editor of the IEE Proceedings on Software Engineering and is currently a TOSEM Associate Editor. He is the co-recipient of the 2005 ACM SIGSOFT Outstanding Research Award for his work in Distributed Software Engineering. He is a Chartered Engineer, Member of the IET and Fellow of the BCS.

Future of Software Engineering(FOSE'07)
0-7695-2829-5/07 $20.00 © 2007 IEEE

Self-Managed Systems: an Architectural Challenge

Jeff Kramer and Jeff Magee
Department of Computing
Imperial College London
SW7 2AZ, UK

{j.kramer, j.magee}@ ic.ac.uk

Abstract

Self-management is put forward as one of the means by which we could provide systems that are scalable, support dynamic composition and rigorous analysis, and are flexible and robust in the presence of change. In this paper, we focus on architectural approaches to self-management, not because the language-level or network-level approaches are uninteresting or less promising, but because we believe that the architectural level seems to provide the required level of abstraction and generality to deal with the challenges posed. A self-managed software architecture is one in which components automatically configure their interaction in a way that is compatible with an overall architectural specification and achieves the goals of the system. The objective is to minimise the degree of explicit management necessary for construction and subsequent evolution whilst preserving the architectural properties implied by its specification. This paper discusses some of the current promising work and presents an outline three-layer reference model as a context in which to articulate some of the main outstanding research challenges.

1 Introduction

As the size, complexity and adaptability required by applications increases, so does the need for software systems which are scalable, support dynamic composition and rigorous analysis, and are flexible and robust in the presence of change. Self-management is the rallying vision. What exactly are self-managed systems? The vision is of systems which are capable of self-configuration, self-adaptation and self-healing, self-monitoring and self-tuning, and so on, often under the flag of self-* or autonomic systems.

For instance, consider that you have a specification of the goals, properties and constraints that you expect your system to achieve and preserve. Consider further that you have a set of software components which implement the required functionality. The aim of self-configuration is that the components should either configure themselves such that they satisfy the specification or be capable of reporting that they cannot.

What if the system suffers from changes in its requirements specification [14] or operational environment such as changes in use, changes in resource availability or faults in the environment or in parts of the system itself? The aim of self-adaptation and self-healing is that the system should reconfigure itself so as to again either satisfy the changed specification and/or environment, or possibly degrade gracefully or report an exception. Change as evolution of the system tends to imply an off-line process in which the system evolves through a number of releases, where each release could employ self-configuration. However, dynamic change, which occurs while the system is operational, is far more demanding and requires that the system evolves dynamically, and that the adaptation occurs at run-time.

Finally, we should note that our required specifications include not only functional behaviour, but also those non-functional properties such as response time, performance, reliability, efficiency and security, and that satisfaction of a specification may well include optimisation.

Clearly this is a challenging vision, which includes almost every one of the research challenges identified in the first FOSE: Software Engineering: A Roadmap at ICSE 2000 [19], namely compositionality, change, NF properties, service-view, perspectives, architecture, configurability, and domain specificity.

Future of Software Engineering(FOSE'07)
0-7695-2829-5/07 $20.00 © 2007 IEEE

How can we approach this dream?

Different research communities are already engaged in relevant research, investigating and proposing approaches to various aspects of self-management for particular domains. For instance, in the networking, distributed systems and services community, there has been the Autonomic Computing conferences [2] and more recently, the SelfMan Workshop 2006 [1] to discuss and analyse the potential of self-* systems for managing and controlling networked systems and services. A special issue associated with SelfMan 2005 on Self-Managed Systems and Services [31] covers a diverse range of descriptions of work on aspects such as self-healing by dynamic fault awareness and self-adaptation/tuning for dealing with transient web overloads. Dobson et al. [17] provide a recent survey on autonomic communications, covering research work on context awareness, autonomic algorithms, trust and security and appropriate adaptation techniques. They propose an autonomic control loop (a phased approach) of actions *collect* (monitoring), *analyse*, *decide* and *act*, a cycle which naturally appears in many proposed approaches. In addition to those mentioned above, there is also the International Conference on Self-Organization and Autonomous Systems in Computing and Communications (SOAS'2006) [3], and the International Conference on Autonomic and Autonomous Systems ICAS 2006 [6].

The proliferation of conferences and workshops mentioned above reflects the interest in the topic; and this is only from the networking, systems and services communities. Other research communities also interested and appropriate include the intelligent agent, machine learning and planning communities, and many others, adopting underlying models as diverse as those derived from biology and social interaction.

In the software engineering community there has been a series of workshops which started in the distributed systems community with the CDS (Configurable Distributed Systems) conferences [4, 9, 5] and more recently with WOSS (Workshop on Self-Healing and Self-Managed Systems) [8, 7] and SEAMS (Software Engineering for Adaptive and Self-Managing Systems) [3]. These conferences and workshops have provided excellent forums for discussing the software issues involved. However, although the work discussed over the years has provided much that is useful in contributing towards self-management, it has not yet resolved some of the general and fundamental issues in order to provide a comprehensive and integrated approach.

Why an architectural approach?

In this paper we focus on the use of an architecture-based approach, as we believe that it offers the following potential benefits:

- *Generality* – the underlying concepts and principles should be applicable to a wide range of application domains, each associated with appropriate software architectures.
- *Level of abstraction* – software architecture can provide an appropriate level of abstraction to describe dynamic change in a system, such as the use of components, bindings and composition, rather than at the algorithmic level.
- *Potential for scalability* – architectures generally support both hierarchical composition and other composition and hiding techniques which are useful for varying the level of description and the ability to build systems of systems, thereby facilitating their use in large-scale complex applications.
- *Builds on existing work* – there is a wealth of architecture description languages and notations which include some support for dynamic architectures and for formal architecture-based analysis and reasoning [12]. These provide a good basis for a rigorous approach which could support evaluation and reasoning, constraints and run-time checks.
- *Potential for an integrated approach* - many ADLs and approaches support software configuration, deployment and reconfiguration. In fact, as mentioned in an accompanying FOSE paper in this proceedings on Software Design and Architecture [37], "software architecture encompasses work in modelling and representation, design methods, analysis, visualization, supporting the realization of designs into code, experience capture and reuse, product lines, deployment and mobility, security, adaptation, and so on."

We are not alone in favouring a component-based architectural approach. Many others also advocate use of architectural principles in their work. For instance, Oreizy et al [34] provide a general outline of an architectural approach which includes adaptation and evolution management; Garlan and Schmerl [21] describe the use of architecture models to support self-healing; Dashofy, van der Hoek and Taylor propose the use of an architecture evolution manager to provide the infrastructure for run-time adaptation and self-

healing in ArchStudio [16]; Gomaa and Hussein [24] describe the use of dynamic software reconfiguration and reconfiguration patterns for software product families; Medvidovic, Rosenblum and Taylor present a language and associated environment for architecture-based development and component evolution [33]; Wang et al [39] describe an experiment to support component-based dynamic software evolution; Baresi et al. [11] suggest the use of contracts expressed as assertions to monitor and check dynamic service compositions in service-oriented systems; and Castaldi et al. [13] extend the concepts of network management to component-based, distributed software systems to propose an infrastructure for both component- and application-level reconfiguration using a hierarchy of managers. Our own work has concentrated on the use of ADLs for software design and implementation from components [29], including limited language support for dynamic change [30], a general model for dynamic change and evolution [28], associated analysis techniques [27] and initial steps towards self-management [23].

In order to try to draw all these threads together, we now propose an architectural reference model as a means of identifying more precisely the concerns and research issues that are needed in progressing towards self-management.

2 Towards an Architectural Model for Self-Management

In taking initial steps in the direction of an architecture model for self-management, we have sought inspiration from the large existing body of work on autonomous systems – namely robotics. The first architectures proposed for self–management correspond nearly exactly with the early sense-plan-act SPA architectures used in robots (cf. autonomic control loop [17] mentioned earlier). For example, Garlan's proposed adaptation framework for self-healing systems [21] consists of monitoring, analysis/resolution and adaptation. The monitoring of system operation corresponds to a robot sensing its environment, the analysis/resolution of faults corresponds to planning and adaptation or the execution of changes corresponds to action in the SPA framework. Indeed, Garlan's framework maintains an abstract model of a system in the same way as SPA robots try to maintain a symbolic model of their environment. It is not surprising that this correspondence exists since a self-managed system is clearly an autonomous system in exactly the same way as a robot is. Both are intended to achieve goals without human intervention. Since the SPA architectures of the early eighties, robot architectures have evolved considerably and now, since the mid-90's, nearly all conform to the three layer architecture described by Gat [22]. In Gat's paper's, the three layers are Control: reactive feedback control, Sequencing: reactive plan execution and Deliberation: planning. In the following, we attempt to interpret this three-level robotic architectural model for self-managed systems. Our goal is to exploit the considerable advances that modern robotic systems have in terms of flexibility and responsiveness over their SPA predecessors.

2.1 Component Control

The bottom layer of Gat's three layer architecture is the control layer. It consists of sensors, actuators and control loops. The bottom layer of a self-managed system consists of the set of interconnected components that accomplish the application function of the system. It must of course include facilities to report the current status of components to higher layers and also include the capability to support component creation, deletion and interconnection. In the same way that the control layer of a robot includes feedback loops to implement primitive behaviours such as wall following and moving to a destination, the bottom layer of a self-managed system will contain behaviours to adjust the operating parameters of components – for example the timeout values in a component implementing a TCP protocol. In summary, this layer of a self-managed system will include self-tuning algorithms, event and status reporting to higher levels and operations to support modification – component addition, deletion and interconnection. An important characteristic of this level, is that when a situation is met that the current configuration of components is not designed to deal with, this layer detects this failure and reports it to higher layers.

2.2 Change Management

The middle layer of Gat's three layer architecture is the sequencing layer which reacts to changes in state reported from the lower levels and executes plans that select new control behaviours and set new operating parameters for existing control layer behaviours. This is reactive plan execution. Given a new situation, this layer executes an action or sequence of actions to handle the new situation. For example, when the robot reaches a target location, this layer will determine what should be done next. In a self-managed system, this layer is responsible for effecting changes to the

underlying component architecture in response to new states reported by that layer or in response to new objectives required of the system introduced from the layer above. This layer can introduce new components; recreate failed components; change component interconnections and change component operating parameters. It consists of a set of plans which are activated in response to changes of the operating state of the underlying system. For example, when a component fails, change management can effect a repair either by changing component connections or by creating new components. In robotic systems, this layer has been implemented in a number of ways from conditional sequencing systems [10] to sets of state machines. Work in the network management area has produced languages such as Ponder [15] which perform a similar function to the planning languages in the context of systems. Ponder is essentially a language which execute actions in response to recognising (possible complex) events. The essential characteristic of this change management layer is that it consists of a set of pre-specified plans which are activated in response to state change from the system below. The layer can respond quickly to new situations by executing what are in essence pre-computed plans. If a situation is reported for which a plan does not exist then this layer must invoke the services of the higher planning layer. In addition, new goals for a system will involve new plans being introduced into this layer.

2.3 Goal Management

The uppermost layer of Gat's three layer architecture is the deliberation layer. This layer consists of time consuming computations such as planning which takes the current state and a specification of a high-level goal and attempts to produce a plan to achieve that goal. An example in robotics would be given the current position of a robot and a map of its environment produce a route plan for execution by the sequencing layer. Changes in the environment, such as obstacles that are not in the map, will involve re-planning. The role of the equivalent layer in a self-managed system is Goal Management. This layer produces change management plans in response to requests from the layer below and in response to the introduction of new goals. For example, if the goal in to maintain some architectural property such as triple redundancy for all servers, this layer could be responsible for finding the resources on which to create new components after failure and producing a plan as how to create and integrate these new components to the change management layer. It could be responsible for deciding the optimal placement of servers for load balancing purposes. As we will address further in the next section there are many research issues here as to how to represent high level system goals, how to synthesize change management plans from these goals and how general or domain specific this layer should be.

Figure 1 summarises our proposed three layer model for a self managed system following Gat's work on architectures for robotic systems. The principal criteria for placing function in different layers in Gat's architecture is one of time scale and this would seem to apply equally well to self managed systems. Immediate feedback actions are at the lowest level and the longest actions requiring deliberation are at the uppermost level. We would emphasize that we do not consider this an implementation architecture but rather a conceptual or reference architecture which identifies the necessary functionality for self management. We will use it in the next section to organise and focus discussion of the research challenges present by self management.

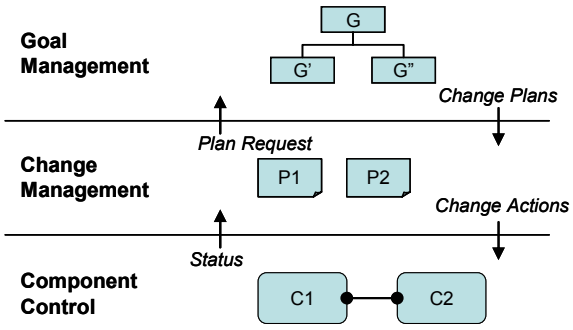


Figure 1 – Three Layer Architecture Model for Self-Management.

3 Research Issues

In the previous section we outlined a three layer architecture model which is intended as a form of reference model rather than as a guide to how self managed software should be implemented. In this section, we use the model to structure the presentation of the research issues we see presented by the challenge of implementing self-managed systems. To ground this discussion, we draw examples from the work with which we are most familiar – namely our own.

3.1 Component Control Layer

We are concerned with management at the architectural level where we consider a system to

consist of a set of interconnected components which may be co-located and/or distributed over a network of communicating computer nodes. Our model of a component is depicted in figure 2 below. A component implements the set of services that it provides and it may use another set of services, denoted the required services, in implementing these services. In addition, a component has an externally visible state which we term its *mode*. The mode is simply an abstracted view of the internal state of a component that is made visible for management purposes. For example, the mode may indicate whether the component is in an active or standby mode. It may in addition indicate non-functional aspects such as the current load on a server. Mode may therefore consist of more than a simple scalar datatype, although in our examples we use only a simple scalar mode. We have used the Darwin [29] format for diagrams updated with modes [25], however, we could equally have used UML 2.0 as the Darwin form of component can now be satisfactorily encoded in UML2.0 [32].

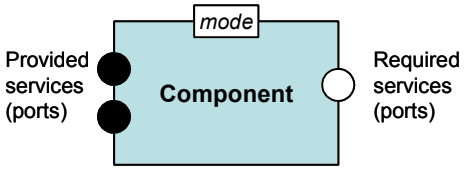


Figure 2 – Example component model

To initially construct and subsequently change systems, we need a set of operations on components. These are typically:

create C: T
– create component instance **C** from type **T**.
delete C
– delete component instance **C**.
bind C_1.r – C_2.p
– connect required port **r** of component **C_1** to provided port **p** of component **C_2**.
unbind C_1.r
– disconnect required port **r** of component **C_1**
set C_1.m to *val*
– set mode **m** of component **C_1** to *val*.

A system constructed in this way will have a configuration or management state consisting precisely of the set of components instances, the set of connections between components and the set component mode values – an example architecture for an autonomous underwater vehicle [20] operating in a mode in which the sonar is passive shown in Figure 3.

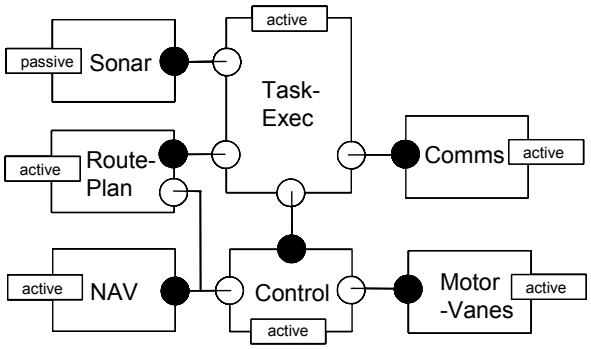


Figure 3 – Example component architecture

The research challenges at this level of a self-managed architecture are primarily concerned with preserving safe application operation during change. For example, the change of mode in a mechatronic system controlling a vehicle [36] can involve moving from one control algorithm to another. The implementation at this level must ensure that the mode change required to adapt to a change of the external operating environment does not generate undesirable transient behavior resulting in, for example, sharp accelerations or decelerations. In systems where the behavior is transactional rather than continuous, the challenge is to ensure that state information is not lost when the configuration is modified. The change management algorithm outlined in [28] tries to ensure stable conditions for change by ensuring that components are passive or quiescent before change. For example, a component can be safely removed from a system if it is isolated (no bindings to or from) and passive (cannot initiate transactions). The challenge is to find scalable algorithms that minimize disruption to the system during change and ensure that system safety properties are not violated. An associated challenge is to verify that safety properties are not violated during change [27], a problem addressed more promisingly by Zhang and Cheng [40].

We have looked at a system as a collection of components; however a component itself may consist of multiple interconnected components. To deal with complex systems in a scalable way, we must deal with hierarchical structure. This raises interesting issues with the respect to the type of a component. When we modify the internal structure of a component, we are

clearly creating a variant of its type. We may want to instantiate new instances of this variant. It is likely that self-management of system will require the online dynamic execution of operations that we currently consider to be offline maintenance or version control operations.

3.2 Change Management Layer

This layer as outlined in section 2 is responsible for executing changes in response either to changes in state reported from the lower layer are in response to goal changes. We think of this layer as a precompiled set of plans or tactics that respond to a predicted class of state change. For example, in a fault-tolerant system, failure of a component may cause a duplicate server to immediately switch from standby to active mode; however, the state change observed by change management should cause the system to create a new standby server. In a fault tolerant system, it is clear that server failure is a predicted state change and the change management layer will include a procedure for dealing with the change. Similarly, we would consider the example repair strategy outlined by Garlan and Schmerl [21] as a plan executed by change management.

One of the major research challenges at this level is dealing with distribution and decentralization. It is this issue of distribution or decentralization that appears to be the essential factor in distinguishing the problem of performing self-management of complex software systems from existing work on robotic systems. Distribution is the most general situation raising issues of latency, concurrency and partial failures, and is likely to be the case (at least for parts of the system) in large and complex applications. Coping with distribution and arbitrary failure leads to the need for some level of local autonomy while preserving global consistency. In essence, distribution contributes the problem of obtaining consistent views of system state on which to base change decisions and decentralization of control brings the problem of robust execution in a situation in which partial failure can occur. Our first attempt to deal with this resulted in a change architecture with completely decentralized change execution; this, however, required state change to be serialized to ensure termination of the configuration in a valid state [23]. This decentralized implementation architecture is shown in Figure 4.

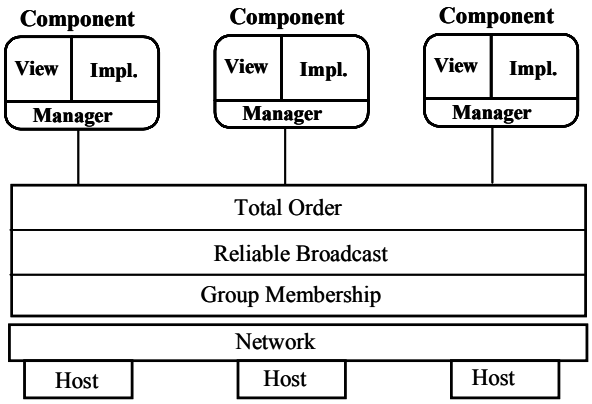


Figure 4 – Decentralized Change Management Implementation architecture

The system depicted in Figure 4, was an experiment to look at extreme distribution in which change management layer functionality was included with each component. Each component maintained a view of the overall system and executed local changes (include connection to other components) in response to state changes in the view. The problems with this system were 1) the view of the system has to be complete and 2) it requires a total order broadcast bus to keep views consistent. Consequently, this was a fully decentralized architecture that reliably executed change in the presence of arbitrary failure. However, it was not a scalable architecture. What we require are systems which can accommodate partial inconsistent views and as a consequence relax the need for totally ordered broadcast communication. The challenge is to find change management algorithms that can tolerate inconsistency and which eventually terminate in a system that satisfies constraints. It is also required that the system does not violate safety constraints while it is converging on a stable state. There are of course examples of self stabilising algorithms [18]; however, these are for specific configurations and applications.

One of the goals of the system described in Figure 4 was to preserve global structural constraints. It was primarily this requirement that dictated a consistent view of system structure. It may well be that taking a more behavioural view of system constraints will provide opportunities for relaxing the consistency requirement. For example, if we are not at all interested in structure, components can simply bind to any service that satisfies the local requirement. Failure of the remote service can trigger a search for a replacement service.

3.3 Goal Management Layer

The initial problem is to have a precise specification of the goals required of a system. These need to encompass both application goals and system goals concerned with self-management. It is likely that the refinement of very high-level goals to precisely specified goals that can be processed by machines will require human assistance as is current practice in goal oriented requirements engineering [38]. The challenge is to achieve goal specification such that it is both comprehensible by human users and machine readable.

If we ignore the problem of precisely specifying structure, behaviour and performance required of a system, the function of goal management can be succinctly described. It takes a declarative specification of system goals, a snapshot of the current state of the system and produces a change plan which moves the system from its current state to a state which satisfies the system goals. Regrettably, this is of course a computationally hard and sometimes intractable problem. Even when tractable, the time taken to generate a plan may not meet the response times required.

Solutions so far have focussed on dealing as far as is possible with planning by designing a set of plans (sometimes referred to as tactics) offline that can be shown either by construction or by a verification process to satisfy system constraints for a range of possible system states. For example, in our system described in [23], we specified system structural constraints in Alloy [26] and developed tactics that could be shown using the Alloy model checker to move a system into a state that satisfied constraints. In other words, the planning problem was done off-line and the problem reduced to one of verification. This approach is sufficient if it can be shown that the set of change plans are sufficient to deal with any possible system state. This is of course exactly what is done in some of the classic fault-tolerant architectures such as active-standby server pairs.

There has been promising work in the autonomous composition of Web services using a "planning as model-checking approach" [35]. In essence, this is in its present state an off-line planning approach. The more challenging problem in the spirit of autonomous self-management is to provide an on-line planner. This is invoked by the change management layer when it finds that none of its current plans apply to the observed system state. This is where the real research challenges in true self-management lie – in the automatic decomposition of goals and in the generation of operationalized plans from these goals. The usual strategy of constraining the problem domain will undoubtedly help.

4 Conclusion

In this paper, we have described our vision of self-management at the architectural level, where a self-managed software architecture is one in which components automatically configure their interaction in a way that is compatible with an overall architectural specification and achieves the goals of the system. We chose to concentrate on an architectural approach as we believe that this offers the required level of abstraction and generality to integrate some of the possible solutions to the challenges posed. We are biased towards a rigorous engineering approach in which low-level actions can be clearly and formally related to high-level goals that are precisely specified.

We have defined a three layer reference model – component control, change management and goal management – to provide a context for discussing the main research challenges which self-management poses. At the component layer, the main challenge is to provide change management which reconfigures the software components, ensures application consistency and avoids undesirable transient behaviour. At the change management layer, decentralized configuration management is required which can tolerate inconsistent views of the system state, but still converge to a satisfactory stable state. Finally, some form of on-line (perhaps constraint based) planning is required at the goal management layer.

To provide a self-managed system, solutions to these challenges need to be integrated to provide a comprehensive solution, supported by an appropriate infrastructure. In addition, the approach must be amenable to a rigorous software development approach and analysis, so as to ensure preservation of desirable properties and avoid undesirable emergent behaviour. A challenge indeed!

References

[1] *2nd IEEE Int. Workshop on Self-Managed Networks, Systems and Services (SelfMan 2006)*, IEEE, Dublin, 2006.

[2] *The 3rd IEEE International Conference on Autonomic Computing* IEEE, Dublin, 2006.

[3] *International Conference on Self-Organization and Autonomous Systems in Computing and Communications (SOAS'2006)*, Erfurt, Germany, September 2006.

0-7695-2829-5/07 $20.00 © 2007 IEEE

[4] *Proceedings of IEE/IFIP 1st Int. Workshop on Configurable Distributed Systems (CDS 92)*, in J. Kramer, ed., London, May 1992.

[5] *Proceedings of IEEE 3rd International Conference on Configurable Distributed Systems (CDS 96)*, in J. Magee and K. Schwan, eds., May 1996.

[6] *Proceedings of International Conference on Autonomic and Autonomous Systems ICAS 2006*, Santa Clara, July 2006.

[7] *Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems*, in D. Garlan, J. Kramer and A. Wolf, eds., ACM Press, Newport Beach, California, 2004, pp. 119.

[8] *Proceedings of the first workshop on Self-healing systems*, in D. Garlan, J. Kramer and A. Wolf, eds., ACM Press, Charleston, South Carolina, 2002, pp. 120.

[9] *Proceedings. of IEEE 2nd International. Conference on Configurable Distributed Systems, Pittsburgh, (CDS 94).* in J. Kramer and J. Purtilo, eds., Pittsburgh, May 1994

[10] C. Agre and D. Chapman, *What are Plans for?*, Robotics and Autonomous Systems, 6 (1990), pp. 17-34.

[11] L. Baresi, C. Ghezzi and S. Guinea, *Smart monitors for composed services*, *Proceedings of the 2nd international conference on Service oriented computing*, ACM Press, New York, NY, USA, 2004.

[12] J. S. Bradbury, J. R. Cordy, J. Dingel and M. Wermelinger, *A survey of self-management in dynamic software architecture specifications*, *Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems*, ACM Press, Newport Beach, California, 2004.

[13] M. Castaldi, A. Carzaniga, P. Inverardi and A. L. Wolf, *A light-weight infrastructure for reconfiguring applications*, *Proceedings of 11th Software Configuration Management Workshop (SCM03)*, LNCS, Portland, Oregon, 2003.

[14] B. H. C. Cheng and J. Atlee, M., *Research Directions in Requirements Engineering*, in L. Briand and A. L. Wolf, eds., *Future of Software Engineering 2007*, IEEE-CS Press, 2007.

[15] N. Damianou, N. Dulay, E. Lupu and M. Sloman, *The Ponder Policy Specification Language*, *Proceedings of the International Workshop on Policies for Distributed Systems and Networks*, Springer-Verlag, 2001.

[16] E. M. Dashofy, A. van der Hoek and R. N. Taylor, *Towards architecture-based self-healing systems*, *Proceedings of the first workshop on Self-healing systems*, ACM Press, Charleston, South Carolina, 2002.

[17] S. Dobson, S. Denazis, Fernndez, Antonio, D. Gati, E. Gelenbe, Massacci, P. Nixon, F. Saffre, N. Schmidt and F. Zambonelli, *A survey of autonomic communications*, ACM Trans. Auton. Adapt. Syst., 1 (2006), pp. 223-259.

[18] S. Dolev, *Self-Stabilization*, MIT Press, 2000.

[19] A. Finkelstein and J. Kramer, *Software engineering: a roadmap*, *Proceedings of the Conference on The Future of Software Engineering*, ACM Press, Limerick, Ireland, 2000.

[20] H. Foster, J. Magee, S. Uchitel and J. Kramer, *Scenario-Based Software Synthesis for Adaptable Software Architectures of UAVs*, *Proceedings of First Annual SEAS DTC Conference*, www.seasdtc.com, Edinburgh, 2006.

[21] D. Garlan and B. Schmerl, *Model-based adaptation for self-healing systems*, *Proceedings of the first workshop on Self-healing systems*, ACM Press, Charleston, South Carolina, 2002.

[22] E. Gat, *Three-layer Architectures*, *Artificial Intelligence and Mobile Robots*, MIT/AAAI Press, 1997.

[23] I. Georgiadis, J. Magee and J. Kramer, *Self-organising software architectures for distributed systems*, *Proceedings of the first workshop on Self-healing systems*, ACM Press, Charleston, South Carolina, 2002.

[24] H. Gomaa and M. Hussein, *Dynamic Software Reconfiguration in Software Product Families*, *5th International Workshop on Software Product-Family Engineering*, LNCS 3014, Springer 2004, 435-444., Siena, Italy, 2003.

[25] D. Hirsch, J. Kramer, J. Magee and S. Uchitel, *Modes for Software Architectures*, *Third European Workshop on Software Architecture (EWSA 2006)*, Springer, Nantes, France, Sept 2006.

[26] D. Jackson, *Software Abstractions: Logic, Language, and Analysis*, MIT Press, 2006.

[27] J. Kramer and J. Magee, *Analysing dynamic change in distributed software architectures*, Software, IEE Proceedings-, 145 (1998), pp. 146-154.

[28] J. Kramer and J. Magee, *The evolving philosophers problem: dynamic change management*, Software Engineering, IEEE Transactions on, 16 (1990), pp. 1293-1306.

[29] J. Magee, N. Dulay, S. Eisenbach and J. Kramer, *Specifying Distributed Software Architectures*, *5th European Software Engineering Conference (ESEC)*, Sitges, Spain, 1995.

[30] J. Magee and J. Kramer, *Dynamic structure in software architectures*, *Proceedings of the 4th ACM SIGSOFT symposium on Foundations of software engineering*, ACM Press, San Francisco, California, United States, 1996.

[31] J.-P. Martin-Flatin, J. Sventek and K. Geihs, *Special Issue on Self-managed systems and services* Commun. ACM, 49 (2006), pp. 36-39.

[32] A. McVeigh, J. Kramer and J. Magee, *Using resemblance to support component reuse and evolution*, *Proceedings of the 2006 conference on Specification and verification of component-based systems*, ACM Press, Portland, Oregon, 2006.

[33] N. Medvidovic, D. S. Rosenblum and R. N. Taylor, *A language and environment for architecture-based software development and evolution*, *Proceedings of the 21st international conference on Software engineering*, IEEE Computer Society Press, Los Angeles, California, United States, 1999.

[34] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimhigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum and A. L. Wolf, *An architecture-based approach to self-adaptive software*, Intelligent Systems and Their Applications, IEEE [see also IEEE Intelligent Systems], 14 (1999), pp. 54-62.

[35] M. Pistore, A. Marconi, P. Bertoli and P. Traverso, *Automated Composition of Web Services by Planning at the Knowledge Level*, *IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence*, Edinburgh, Scotland, 2005.

[36] W. Schaefer and H. Wehrheim, *The Challenges of Building Advanced Mechatronic Systems*, in L. Briand and A. L. Wolf, eds., *Future of Software Engineering 2007*, IEEE-CS Press, 2007.

[37] R. N. Taylor and A. van der Hoek, *Software Design and Architecture: The Once and Future Focus of Software Engineering*, in L. Briand and A. L. Wolf, eds., *Future of Software Engineering 2007*, IEEE-CS Press, 2007.

[38] A. van Lamsweerde, *Goal-Oriented Requirements Engineering: A Guided Tour*, *Proceedings of the 5th IEEE International Symposium on Requirements Engineering*, IEEE Computer Society, 2001.

[39] Q. Wang, J. Shen, X. Wang and H. Mei, *A component-based approach to online software evolution: Research Articles*, J. Softw. Maint. Evol., 18 (2006), pp. 181-205.

[40] J. Zhang and B. H. C. Cheng, *Model-based development of dynamically adaptive software*, *Proceeding of the 28th international conference on Software engineering*, ACM Press, Shanghai, China, 2006.

Architecture-Based Self-Protecting Software Systems

Eric Yuan, Sam Malek
George Mason University
4400 University Drive
Fairfax, VA, 22030
{eyuan, smalek}@gmu.edu

Bradley Schmerl, David Garlan, Jeff Gennari
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA, 15213
{schmerl, garlan, jgennari}@cs.cmu.edu

ABSTRACT

Since conventional software security approaches are often manually developed and statically deployed, they are no longer sufficient against today's sophisticated and evolving cyber security threats. This has motivated the development of *self-protecting software* that is capable of detecting security threats and mitigating them through runtime adaptation techniques. In this paper, we argue for an *architecture-based self- protection (ABSP)* approach to address this challenge. In ABSP, detection and mitigation of security threats are informed by an architectural representation of the running system, maintained at runtime. With this approach, it is possible to reason about the impact of a potential security breach on the system, assess the overall security posture of the system, and achieve defense in depth. To illustrate the effectiveness of this approach, we present several architecture adaptation patterns that provide reusable detection and mitigation strategies against well-known web application security threats. Finally, we describe our ongoing work in realizing these patterns on top of Rainbow, an existing architecture-based adaptation framework.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures

Keywords

Self-Protection; Software Architecture; Software Security

1. INTRODUCTION

Security is increasingly the principal concern that drives the design and construction of modern software systems. Conventional, often manually developed and statically employed, techniques for securing software systems are no longer sufficient. As adversaries become more agile in devising new threats, so should the mechanisms for securing the software systems. This has motivated the development of *self-protecting software*—a new kind of software, capable of detecting security threats and mitigating them through runtime adaptation techniques.

Most self-protection research to-date has focused on specific line(s) of defense (e.g., network, host, or middleware) within a software system. Specifically, such approaches tend to focus on a specific type or category of threats, implement a single strategy or technique, and/or protect a particular component or layer of the system [31]. As a result, the "big picture" understanding of overall security posture and globally coordinated defense strategies appear to be lacking. In addition, growing threats of insider attack call for new mechanisms to complement traditional perimeter-based security (i.e., securing the system at its boundaries) that has been the main focus of prior research. Finally, due to the added complexity of dynamically detecting and mitigating security threats, the construction of self-protecting software systems has shown to be significantly more challenging than traditional software [7]. Lack of engineering principles and repeatable methods for the construction of such software systems has been a major hindrance to their realization and adoption by industry.

In this paper, we argue for an *architecture-based self-protection (ABSP)* approach to address the aforementioned challenges. In ABSP, detection and mitigation of security threats are informed by an architectural representation of the software that is kept in synch with the running system. An architectural focus enables the approach to assess the overall security posture of the system and to achieve defense in depth, as opposed to point solutions that operate at the perimeters. By representing the internal dependencies among the system's constituents, ABSP provides the means to tackle issues such as insider attack. The architectural representation also allows the system to reason about the impact of a security breach on the system, which would inform the recovery process.

As concrete evidence of how ABSP promotes a disciplined and repeatable process for engineering self-protecting software systems, we will present several *architecture-level self-protection patterns*. These patterns provide reusable detection and adaptation strategies for solving well-known security threats. We illustrate their application in dealing with commonly encountered security threats in the realm of web-based applications.

Finally, we describe our work in realizing some of these patterns on top of an existing architecture-based adaptation framework, namely Rainbow [9]. The resulting framework holds promise for increasing reuse across applications/domains and reducing the effort required in realizing self-

QoSA'13, June 17–21, 2013, Vancouver, BC, Canada.

protection capabilities. In the process of extending Rainbow, we have faced several challenging research questions that will frame future research in this area.

The remainder of paper is organized as follows: in Section 2 we set the stage with a simple web application as a motivating example, so that we can illustrate in a concrete setting how ABSP can help thwart security attacks. In Section 3 we briefly outline related self-protection research and point out their limitations, before making the case for the architecture-based approach in Section 4. In section 5 we proceed to elaborate the details of ABSP using a number of repeatable architecture adaptation patterns. In Section 6 we provide an overview of our ongoing work in implementing the security adaptation patterns in the Rainbow framework. Finally, we conclude the paper with a summary of our key contributions, a discussion of remaining challenges, and areas of future research.

2. MOTIVATING EXAMPLE

Based on real sites like cnn.com, Znn [6] is a news service that serves multimedia news content to its customers. Architecturally, Znn is a web-based client-server system that conforms to an N-tier style. As illustrated in Figure 1, the service provides web browser-based access to a set of clients. To manage a high number of client requests, the site uses a load balancer to balance requests across a pool of replicated servers (two shown), the size of which can be configured to balance server utilization against service response time. For simplicity sake we assume news content is retrieved from a database and we are not concerned with how they are populated. We further assume all user requests are stateless HTTP requests and there are no transactions that span across multiple HTTP requests. This base system does not yet have any architectural adaptation features, but serves as a good starting point for our later discussions.

To illustrate the ABSP approach as well as self-adaptation concepts, we augment Znn's basic web application architecture with a "meta component" called ARchitecture Manager (ARM), shown in Figure 1, which is responsible for monitoring the components and connectors in the system and adapting them in accordance with self-protection goals. To achieve this, the ARM implements the Monitor, Analyze, Plan, Execute (MAPE) loop [14], and is connected to "sensors" throughout the system to monitor their current status and "effectors" that can receive adaptation commands to make architecture changes to the base system at runtime. As described later in the paper, the Rainbow platform [9] serves as a good implementation of the ARM component using architecture-based techniques combined with control and utility theories.

In the real world, a public website like Znn faces a wide variety of security threats. Malicious attacks, both external and internal, grow more complex and sophisticated by the day. The Open Web Application Security Project (OWASP) maintains a Top Ten list [22] that provides a concise summary of what was considered to be the ten most critical security risks at the application level. They are listed in Table 1. Another list published by the MITRE Corporation, the Top 25 Common Weakness Enumeration (CWE) [26], covers many similar threats with more details.

We will use Znn throughout the paper as a running example to illustrate how architecture-based approaches may be utilized to help counter such threats in an autonomous and self-managed fashion. Note that we focus our attention on the OWASP Top 10 and CWE Top 25 threats due to their prevalence on the Internet and their relevance to the Znn web application. However the same general approach can be applied to counter other security threats not covered in the two lists, such as Denial of Service attacks, buffer overflows, privilege escalation, etc.

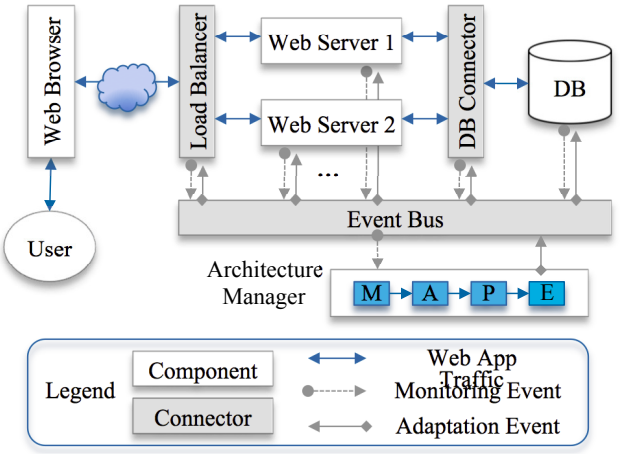


Figure 1: Znn Self-Adaptive Architecture

Table 1: OWASP Top 10, 2010 Edition

A1	Injection
A2	Cross-Site Scripting (XSS)
A3	Broken Authentication and Session Management
A4	Insecure Direct Object References
A5	Cross-Site Request Forgery (CSRF)
A6	Security Misconfiguration
A7	Insecure Cryptographic Storage
A8	Failure to Restrict URL Access
A9	Insufficient Transport Layer Protection
A10	Unvalidated Redirects and Forwards

3. STATE OF THE ART

In SEAMS 2012 the authors presented a preliminary study of self-protecting software systems [31]. We have since expanded that study following a systematic literature review process proposed by [15]. This has broadened the scope of our study from 32 to 107 papers and strengthened the validity of our conclusions. To the best of our knowledge, this study [32] is the most comprehensive investigation of the literature in this area of research.

One of the most evident findings in our survey is that most self-protection research to-date focuses on specific layers or "lines of defense" in a software system, namely:

- *Network*, involving security of communication links, networking protocols, and data packets.
- *Host*, involving the host environment on a machine, including hardware/firmware, OS, and in some occasions hypervisors that support virtual machines.
- *Middleware* such as application servers, object brokers, messaging hubs and service buses.
- *Application* level, concerning programming language security and application-specific measures.

Such research, no matter how effective, tends to be point solutions that lack the overall picture of the system's security posture. For example, the business context in which the system is deployed and inter-component collaboration to help defend against security breaches are not considered by these solutions. On the other hand, self-adaptive approaches that focus on software architecture as a whole (and as such can be applied to any/all component layers in a coordinated and consistent fashion), can provide a context for reasoning about these business- and application-specific concerns.

A related finding in our survey also showed that self-protection research has been predominantly perimeter- focused, aiming to protect the system at its boundary. The historical emphasis of the computer security community on network intrusion detection and intrusion prevention (ID/IP) helps to explain this. Even though we observed encouraging trends from intrusion detection systems to intrusion tolerant systems (ITS) [18] and automated intrusion response systems (IRS) [25] in the past decade, the continued influence of the intrusion-centric mindset is evident. Systems relying solely on perimeter security, however, are often rendered helpless when the perimeter is breached; nor can they effectively deal with threats that originate from inside of the system. With growing threats of insider attacks [23], we see a pressing need for architectural approaches that monitor and adapt overall system behavior.

Of equal relevance to this paper are the recurring self-protection patterns revealed in the same survey. Architecture and design patterns in general have been well-researched and documented, but we observed a number of interesting patterns have emerged as being especially effective in establishing self-protecting behavior. We identified a number of *structural patterns* and *behavioral patterns*, a subset of which will be applied to ABSP in Section 5.

4. THE CASE FOR ARCHITECTURE-BASED SELF-PROTECTION

Mature and effective defense mechanisms are readily available to make the Znn news site more secure. At the network and host levels, for example, one can:

- Place a traffic-filtering firewall at the WAN edge before the load balancer to filter all incoming TCP/IP traffic. Illegal access to certain ports and protocols from certain source IP addresses can be easily blocked.
- Install a network-based intrusion detection device on the local LAN that examines network traffic to detect abnormal patterns. Repeated requests to a single server that exceed a threshold, for example, may trigger an alarm for Denial of Service (DoS) attack.
- Install a host-based intrusion detection system in the form of software agents on all servers, which monitors system calls, application logs, and other resources. Access to a system password file without administrator permissions, for instance, may indicate the server has been compromised.

At the web application level, OWASP has also recommended a rich set of detection techniques and prevention countermeasures aimed at mitigating the aforementioned Top 10 risks. Some key practices include [22]:

- Conduct code reviews and blackbox penetration testing to find security flaws proactively;
- Employ static and dynamic program analysis tools to identify application vulnerabilities. For SQL injection risks, for instance, one can conduct static code analysis to find all occurrence of the use of SQL statement interpreters;
- Use whitelist input validation to ensure all special characters in form inputs are properly escaped to prevent Cross-Site Scripting (XSS) attacks;
- Follow good coding practices and use well-tested API libraries.

The traditional approaches, however, are not without limitations. First, most are labor-intensive and require significant manual effort during development and/or at runtime. Second, the testing tools and preventive countermeasures do not provide complete and accurate defense against rapidly changing attack vectors, as admitted in the OWASP Top 10 report. Many approaches find it difficult to balance the competing goals of maximizing detection coverage (reducing false negatives) and maintaining detection accuracy (reducing false positives). Furthermore, for web attacks such as XSS and CSRF, they are partly caused by careless and unwitting user behavior, which is impossible to completely eliminate. Last but not the least, a large percentage of the web applications today are so-called "legacy" applications for which development had ended some time ago. Even when vulnerabilities are found, the fixes are going to be difficult and costly to implement.

The Architecture-based Self-Protection (ABSP) approach does not seek to replace the mainstream security approaches but rather to complement them. ABSP focuses on securing the architecture as a whole, as opposed to specific components such as networks or servers. Working primarily with constructs such as components, connectors and architecture styles, the ABSP approach protects the system against security threats by (a) modeling the system using machine-understandable representations, (b) incorporating security objectives as part of the system's architectural properties that can be monitored and reasoned with, and (c) making use of autonomous computing principles and techniques to dynamically adapt the system at runtime in response to security threats, without necessarily modifying any of the individual components.

The Benefits of ABSP are as follows:

- Defense in depth. Most self-protection approaches that have originated in the systems community are perimeter based. By modeling the internal structure of a software system in ABSP, it provides an effective mechanism to deal with threats in multiple stages and at different levels.
- Impact Analysis. ABSP allows us to reason about threat impacts as well as the trustworthiness of a software system at the granularity of its elements (components, connectors). The dependencies among the system's constituents would help us track the impact of a compromised element on the other parts of the system and formulate globally coordinated defense strategies.
- Insider attack. Modeling and monitoring the software system in terms of its architectural constituents make it possible to detect abnormal behavior inside the software system. This allows us to mitigate security threats that originate from within the system.

- Reuse. By implementing self-protection patterns as orthogonal concerns, separate from application logic, the former may be easily reused in other applications.
- Dynamism. Separation of concerns also allows the self-protection mechanisms to evolve independent of the application logic, to quickly adapt to emerging threats.

5. ARCHITECTURE PATTERNS FOR SELF-PROTECTION

In this section, we use patterns as a convenient way to illustrate how ABSP may be used to bring self-securing capabilities to a system such as Znn. For each pattern, we briefly describe the security threat(s) it can be effective for, how the threat could be detected, and finally how it could be dealt with through adaptation. It is worth noting that the self-protection patterns described here represent architectural level adaptation strategies, and are therefore different from previously identified reusable security patterns [12, 30], which constitute for the most part implementation tactics such as authentication, authorization, password synchronization, etc.

5.1 Protective Wrapper Pattern

This simple pattern involves placing a security enforcement proxy, wrapper, or container around the protected resource, so that request to / response from the resource may be monitored and sanitized in a manner transparent to the resource. Protective wrappers are not uncommon in past self-protection research. The SITAR system [29] protects COTS servers by deploying an adaptive proxy server in the front, which detects and reacts to intrusions. Invalid requests/responses trigger reconfiguration of the COTS server. Virtualization techniques are increasingly being used as an effective protective wrapper platform. VASP [33], for example, is a hypervisor-based monitor that provides a trusted execution environment to monitor various malicious behaviors in the OS.

5.1.1 Architectural Adaptation

One straightforward way to employ this pattern for Znn is to place a new connector called "Application Guard" in front of the load balancer, as shown in Figure 2. To support this change, the ARM not only needs to connect to the application guard via the event bus, but also needs to update its architecture model to define additional monitoring events for this new element (e.g. suspicious content alerts) and additional effector mechanisms for its adaptation (e.g. blocking a user).

5.1.2 Threat Detection

The application guard serves as a protective wrapper for the Znn web servers by performing two key functions: attack detection and policy enforcement. By inspecting and if necessary sanitizing the incoming HTTP requests, the application guard can detect and sometimes eliminate the threats before they reach the web servers. Injection attacks (OWASP A1), for example, often contain special characters such as single quotes which will cause erroneous behavior in the backend database when the assembled SQL statement is executed. By performing input validation (e.g. using a "white list" of allowed characters) or using proper escape routines, the wrapper can thwart many injection attempts.

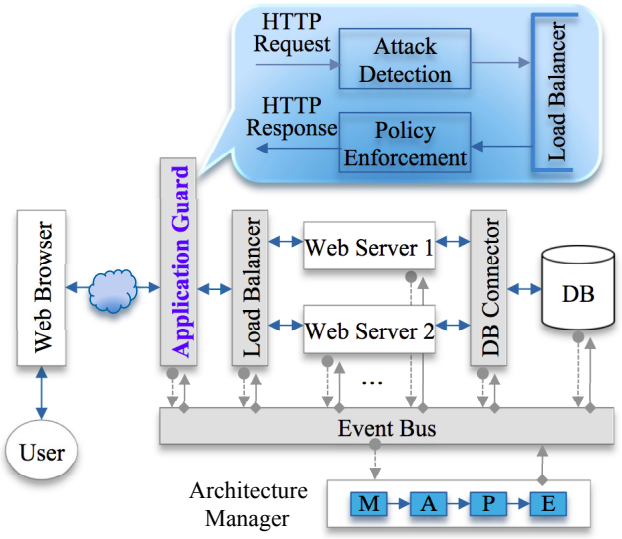


Figure 2: Znn Protective Wrapper Architecture

As its name suggests, this protective wrapper works at the application level, in contrast to conventional network firewalls that focus on TCP/IP traffic. It communicates with and is controlled by the model-driven ARM "brain", and as such can help detect more sophisticated attack sequences that are multi-step and cross-component. For example, the ARM can signal the application guard to flag and stop all access requests to the web server document root, because a sensor detected a buffer overflow event from the system log of the web server host. The latter may have compromised the web server and placed illegitimate information at the document root (e.g., a defaced homepage, or confidential user information). The detection may be achieved through incorporating an attack graph in the ARM's architecture model, as described in [8].

5.1.3 Threat Mitigation

A second function performed by the application guard is policy enforcement as directed by the ARM. Take Broken Authentication and Session Management (OWASP A3) for example; web sites often use URL rewriting which puts session IDs in the URL: `http://znn.com/premium/newsitem;sessionid=SIRU3847YN9W38475N?newsid=43538` When this URL is either shared or stolen, others will be able to hijack the session ID to access this user's content or even his/her personal information. The application guard can easily prevent this by applying encryption / obfuscation techniques so session IDs cannot be identified from the URL, or by tying session IDs with user's MAC addresses so that session IDs cannot be reused even if they are stolen. Similar mechanisms at the application guard may also help patch up other vulnerabilities such as Insecure Direct Object References (OWASP A4) and Failure to Restrict URL Access (OWASP A8).

More adaptive enforcement actions are possible thanks to the ARM component that is aware of the overall system security posture. After the ARM senses the system is under attack, it can instruct the application guard to dynamically cut off access to a compromised server, switch to a stronger encryption method, or adjust trustworthiness of

certain users. Adaptation strategies may be based on heuristic metrics indicating the overall system's security posture, which are computed in real time. As a concrete example, in section 6 we will show how this pattern is employed against denial of service (DoS) attacks.

5.2 Software Rejuvenation Pattern

As defined by [13], the Software Rejuvenation pattern involves gracefully terminating an application instance and restarting it at a clean internal state. This pattern is part of a growing trend of proactive security strategies that have gained ground in recent years. By proactively "reviving" the system to its "known good" state, one can limit the damage of undetected attacks, though at the cost of extra hardware resources. The TALENT system [20], for example, addresses software security and survivability using a "cyber moving target" approach, which proactively migrates running applications across different platforms on a periodic basis while preserving application state. The Self-Cleansing Intrusion Tolerance (SCIT) architecture [17] uses redundant and diverse servers to periodically "self-cleanse" the system to pristine state. the Proactive Resilience Wormhole (PRW) effort [24] also employs proactive rejuvenation for intrusion tolerance and high availability.

5.2.1 Architectural Adaptation

When applying this pattern to the Znn application, we will update the ARM's system representation to establish two logical pools of web servers: in addition to the active server pool connected to the load balancer, there will also be an idle web server pool containing running web servers in their pristine state, as shown in Figure 3. These server instances could be either separate software processes or virtual machine instances. In the simplest case, the ARM will issue rejuvenation commands at regular intervals (e.g., every 5 minutes, triggered by a timer event), which activate a new web server instance from the idle pool and connects it to the load-balancer. At the same time, an instance from the active pool will stop receiving new user requests and terminate gracefully after all its current user sessions log out or time out. The instance will then be restarted and returned to the idle pool.

The ARM "brain" may also pursue more complex rejuvenation strategies, such as:

- Use threat levels or other architectural properties to determine and adjust rejuvenation intervals at runtime
- Perform dynamic reconfigurations and optimizations (e.g. restart a server instance with more memory based on recent server load metrics)
- Mix diverse web server implementations (e.g. Apache and Microsoft IIS) to thwart attacks exploiting platform-specific vulnerabilities

Note that, the rejuvenation process, short as it may be, temporarily reduces system reliability. Extra care is needed to preserve application state and transition applications to a rejuvenated replica.

5.2.2 Threat Detection

A unique characteristic about the rejuvenation pattern is that it neither helps nor is dependent upon threat detection, but for the most part used as a mitigation technique.

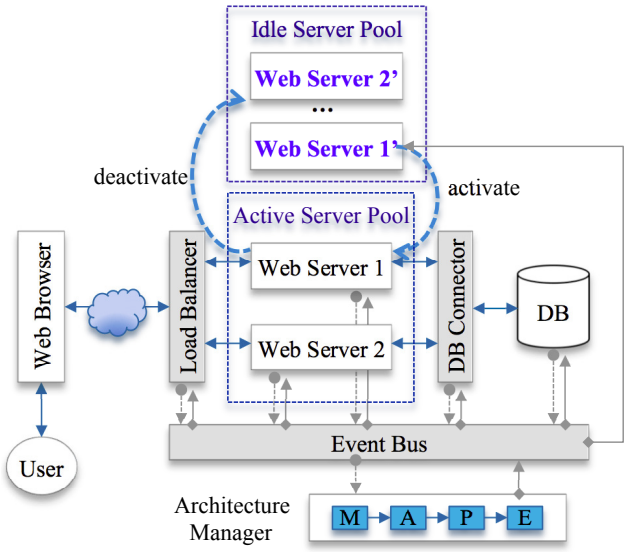


Figure 3: Znn Software Rejuvenation Architecture

5.2.3 Threat Mitigation

Although the rejuvenation pattern doesn't eradicate the underlying vulnerability, it can effectively limit potential damage and restore system integrity under injection (OWASP A1), reflective XSS (OWASP A2), and to some extent CSRF (OWASP A5) attacks, detected or undetected. These are considered among the most vicious and rampant of web application threats, in part because the attack vector is often assisted by careless or unsuspecting users. Clicking on a phishing URL is just one of the many examples. When a fragment of malicious code is sent to the server, such as the following that steals user cookies [21]:

```
<SCRIPT type="text/javascript">
var adr = 'example.com/evil.php?cakemonster='
          + escape (document.cookie);
</SCRIPT>
```

This piece of injected code may be stored in server memory or (in a worse case) in the database, and then used for malicious intents such as stealing confidential user information, hijacking the server to serve up malware, or even defacing the website - and continue doing so *as long as the server is running.*

With a rejuvenation pattern in place, a server may only be compromised for up to the rejuvenation interval. In mission-critical operations, the interval can be as short as a few seconds, drastically reducing the probability and potential damage from these attacks even when detection sensors fail.

Our pattern implementation as depicted in Figure 3 does have some limitations. First, for persistent attacks such as DoS, rejuvenating the web server will not be effective because the DoS traffic will simply be directed to the new web server instance and overwhelm it. In such cases rejuvenation must be carried out in conjunction with other countermeasures such as blocking the attacking source. Secondly, caution must be taken so that corrupted state is not migrated to the new instances. For example, when malicious code is stored in the database, simply recycling the web server will not eradicate the root of the threat because restarting a database server instance will only clean up transient, in-

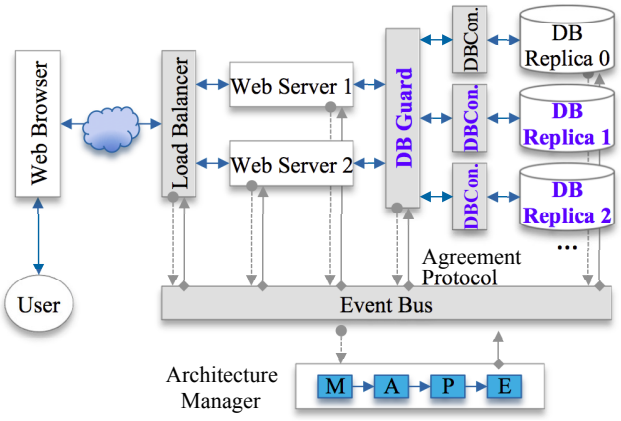


Figure 4: Znn Agreement-Based Redundancy Architecture

memory storage but has no effect on data changes already committed to permanent storage. This pattern, therefore, is not an effective mechanism against stored XSS attacks.

5.3 Agreement-based Redundancy

As pointed out in the previous subsection, proactively "hot swapping" active and possibly tainted web servers with new pristine instances can effectively limit the damage of scripting attacks that seek to inject malicious code in the web server, but the technique can offer little relief to attacks that have succeeded in permanently altering the system state, particularly in the database. To address the latter challenge, we consider another architecture pattern, Agreement-based Redundancy, which maintains multiple replica of a software component at runtime in order to tolerate Byzantine faults and ensure correctness of results. A prime example of this pattern comes from the seminal work by [4] described a Byzantine Fault Tolerance (BFT) algorithm that can effectively tolerate f faulty nodes with $3f+1$ replicas within a short window of vulnerability. Similar agreement-based voting protocols have been used in many other systems such as SITAR and [28]. The strengths of this pattern is many-fold - it is easy to implement, performs well, helps meet both system security and availability goals, and is effective against unknown attacks.

5.3.1 Architectural Adaptation

In the Znn example we choose to apply this pattern to the database layer, as shown in Figure 4. First, we update the ARM's architecture representation to maintain a number of identical database instances (along with their respective database connectors), all active and running concurrently. Secondly, a new connector called DB Guard is introduced to handle database requests from web servers using an agreement-based protocol. The ARM communicates the agreement-based protocol specifics to the DB Guard, such as the number of replicas and quorum thresholds. The ARM can dynamically adapt the protocol as needed at runtime.

5.3.2 Threat Detection

Given the heavy reliance on databases in today's software applications, it is no surprise SQL injection is ranked as the number one threat in both OWASP Top 10 and CWE Top 25. The ABR pattern can effectively detect and stop the SQL variants of the injection attack (OWASP A1) and stored XSS (OWASP A2) attack when they contain illegal writes to the Znn database. Consider a simplified scenario where the Znn web server attempts to retrieve news articles from a database table based on keyword:

```
...
string kw = request.getParameter("keyword");
string query = "SELECT * FROM my_news
  WHERE keyword = '" + kw + "'";
...
```

Note that many database servers allow multiple SQL statements separated by semicolons to be executed together. If the attacker enters the following string:

```
xyz'; DELETE FROM news; --
```

Then two valid SQL statements will be created (note the training pair of hyphens will result in the trailing single quote being treated as a comment thus avoiding generating a syntax error, see [27] for details):

```
SELECT * FROM my_news WHERE keyword='xyz';
DELETE FROM news;
```

As a result, a seemingly harmless database query could be used to corrupt the database and result in loss of data. Good design and coding techniques, along with static analysis tools can help identify vulnerabilities. As mentioned earlier in the paper, however, such efforts are labor-intensive and not bullet-proof. Using the Agreement-based Redundancy pattern, we take an non-intrusive approach that does not require code changes to the web application nor the database. Instead. we execute the following algorithm in the DB Guard connector:

1. Treat each database request R as a potential fault-inducing operation, and execute it first on the primary node (replica 0). The selection of the primary node is arbitrary.
2. Use a voting algorithm to check predefined properties of the database. For example, one property may be the number of news articles. The ARM is responsible for defining and monitoring these properties and making them available to the database connector. When quorums can be reached on all properties and the primary node is part of the quorum, proceed to next step; otherwise flag R as invalid and revert the primary node to its state before R, either by rolling back the transaction or by making a copy of another replica.
3. Execute R on all other replicas 1 to n, bringing all replicas to the same state.
4. Adjudicate the results from all replicas using the voting algorithm. If a quorum is reached, return the result to client; otherwise consider the system in a compromised state and raise flag for human administrator attention.

It is easy to see that when the above attack string gets sent to the DB Guard and executed in the primary node, the number of news articles is reduced to zero after the delete command, therefore different from the quorum. The request will be aborted and the system reverted to its valid state. Our implementation, however, comes with a caveat: it is not effective when the SQL injection seeks only to read data (i.e. the compromise is in system confidentiality, not integrity). In the latter case, the protective wrapper pattern can still help inspect and detect the anomaly.

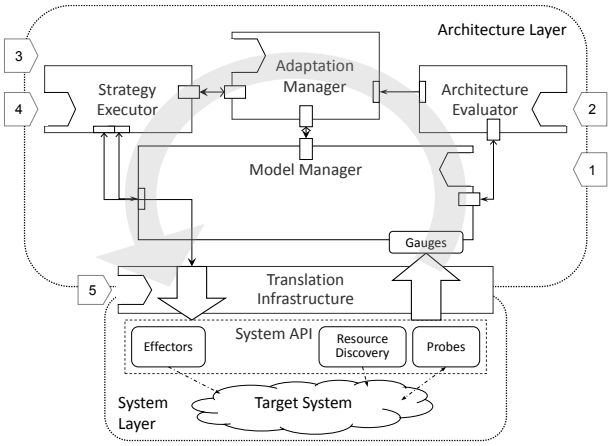


Figure 5: The Rainbow Architecture.

5.3.3 Threat Mitigation

As we have seen from the above scenario, the SQL injection attack is effectively stopped after it is detected in the algorithm. To complete the full sequence for threat mitigation, we only need to furbish a few more details:

- Once an invalid and potentially malicious request is detected, the DB Guard will notify the ARM that can deploy countermeasures such as nullifying the associated user session, notifying the system administrator, or even disabling the user account in question.
- When the adjudication of the results (step 4 of the algorithm) is not unanimous, raise a flag about the minority server instance. If further diagnostics confirm the instance is not in a valid state, destroy and regenerate this instance.

6. REALIZING SELF-PROTECTION PATTERNS IN RAINBOW

In the previous sections, we have argued that architecture-based self-protection can provide a principled and repeatable approach to constructing self-protecting systems, and given some examples of patterns for ABSP. In this section, we outline how to implement this approach in an architecture-based self-adaptive framework called Rainbow. We begin by providing an overview of Rainbow, and then continue by discussing how the Protective Wrapper pattern can be realized by the framework.

6.1 Rainbow Framework Overview

The Rainbow framework has demonstrated how architecture models of the system, updated at runtime, can form the basis for effective and scalable problem detection and correction. Architecture models represent a system in terms of its high level components and their interactions (e.g., clients, servers, data stores, etc.), thereby reducing the complexity of those models, and providing systemic views on their structure and behavior (e.g., performance, availability, protocols of interaction, etc.). In the context of this paper, the Rainbow framework can be viewed as Architecture Manager capable of evaluating and adapting the underlying system to defend against threats.

The Rainbow framework uses software architectures and a reusable infrastructure to support self-adaptation of software systems. Figure 5 shows the adaptation control loop of Rainbow. Probes are used to extract information from the target system that update the architecture model via Gauges, which abstract and aggregate low-level information to detect architecture-relevant events and properties. The architecture evaluator checks for satisfaction of constraints in the model and triggers adaptation if any violation is found, i.e., an adaptation condition is satisfied. The adaptation manager, on receiving the adaptation trigger, chooses the "best" strategy to execute, and passes it on to the strategy executor, which executes the strategy on the target system via effectors.

The best strategy is chosen on the basis of stakeholder utility preferences and the current state of the system, as reflected in the architecture model. The underlying decision making model is based on decision theory and utility [19]; varying the utility preferences allows the adaptation engineer to affect which strategy is selected. Each strategy, which is written using the Stitch adaptation language [5], is a multi-step pattern of adaptations in which each step evaluates a set of condition-action pairs and executes an action, namely a tactic, on the target system with variable execution time. A tactic defines an action, packaged as a sequence of commands (operators). It specifies conditions of applicability, expected effect and cost-benefit attributes to relate its impact on the quality dimensions. Operators are basic commands provided by the target system.

As a framework, Rainbow can be customized to support self-adaptation for a wide variety of system types. Customization points are indicated by the cut-outs on the side of the architecture layer in Figure 5. Different architectures (and architecture styles), strategies, utilities, operators, and constraints on the system may all be changed to make Rainbow reusable in a variety of situations. In addition to providing an engineering basis for creating self-adapting systems, Rainbow also provides a basis for their analysis. By separating concerns, and formalizing the basis for adaptive actions, it is possible to reason about fault detection, diagnosis, and repair. For example, many of the standard metrics associated with classical control systems can, in principle, be carried over: settling time, convergence, overshoot, etc. In addition, the focus on utility as a basis for repair selection provides a formal platform for principled understanding of the effects of repair strategies.

In summary, Rainbow uses architectural models of a software system as the basis for reasoning about whether the system is operating within an acceptable envelope. If this is not the case, Rainbow chooses appropriate adaptation strategies to return the system to an acceptable operating range. The key concepts of the approach are thus: (a) the use of abstract architectural models representing the runtime structures of a system, that make reasoning about system-wide properties tractable, (b) detection mechanisms that identify the existence and source of problems at an architectural level, (c) a strategy definition language called Stitch that allows architects to define adaptations that can be applied to a system at runtime, and (d) a means to choose appropriate strategies to fix problems, taking into consideration multiple quality concerns to achieve an optimal balance among all desired properties.

6.2 Realizing the Protective Wrapper Pattern

In this section, we describe how the Protective Wrapper Pattern in Section 5.1 is implemented in Rainbow to protect Znn against a denial of service (DoS) attack. DoS has been extensively researched in the past, including recent efforts using adaptive approaches [1, 16]. Our focus in this paper is not so much on advancing the state of the art for DoS attack mitigation, but on illustrating how the problem may be addressed at the architectural level using repeatable patterns and a runtime self-adaptive framework.

6.2.1 Architecture Adaptation

At the system level, the protective wrapper is placed in front of the load balancer of Znn to achieve two levels of protection: 1) it maintains a list of black-listed IPs that are considered to be malicious, and ignores all requests from them; and 2) it maintains a list of suspicious clients that are generating an unusually large amount of traffic and limits the requests that get forwarded to the load balancer. Each of these are manipulated via effectors in Znn (in reality, scripts that can run on the load balancer) that introduce and configure the wrapper. Each script is associated with an architectural operator that can be used by tactics in Stitch to implement the mitigation.

The architecture model of Znn is annotated with properties to determine the request rates of clients, and the probability that a client is engaging a DoS (i.e., being malicious). Gauges report values for these properties (described below), and constraints check that clients have reasonable request rates and low probabilities of maliciousness, and if not, are throttled or on the blacklist. If these constraints fail, then the mitigation strategy above is applied.

In terms of customization of Rainbow, the model and its annotation with the above properties corresponds to customization point 1 in Figure 5, and the constraints that check the correctness of the model to point 2.

6.2.2 Threat Detection

The DoS attack is detected by probes that monitor the traffic in the system, and correspond partially to customization point 5 in Figure 5. Rainbow aggregates this data into actionable information within gauges, and then uses this information to update the architectural model to reflect operating conditions. To determine the probability of a client participating in a DoS, we follow the approach described in [3, 2]. We define transactions representing request behaviors in the architectures that are derived from low level system events, and an Oracle that analyzes the transactions from each client and, using a method called Spectrum-based Fault Multiple Fault Localization, reports the probability of each client acting suspiciously as a *maliciousness* property on each component in the model.

Furthermore, probes and gauges keep track of which clients are in the blacklist or being throttled, allowing the constraints in the model to fail only on clients that haven't been dealt with yet.

6.2.3 Threat Mitigation

When a threat is detected and reported in the architectural model, causing a constraint to fail, the Rainbow Adaptation Manager is invoked. It selects and executes adaptations to maximize the utility of the system. In the case of a DoS attack, maximizing utility means stopping the attack with minimal client service disruption. That is, the DoS response must take care not deny access to clients without cause.

Consider the scenario where attackers may be dealt with differently depending on the frequency with which they attack (i.e. repeat offenders) and the duration of the attack. Rules to determine how these factors influence response could be encoded into a security policy with the following logic:

- The traffic for previously unknown attackers is throttled (i.e., some requests are ignored) to limit the impact of the attack without totally cutting off service. This approach minimizes the chances of disrupting the service of possibly legitimate clients.
- Repeat offenders are *blackholed* meaning all that client's traffic is filtered at the load balancer. Known malicious clients are given less mercy than those who have not previously attacked Znn.
- Long-running attacks are not tolerated under any circumstances.

These rules are encoded as the Rainbow strategy shown in Listing 1. The applicability of the FixDosAttack strategy depends on the state of the system as it is represented in the architectural model. The strategy is only applicable if the cUnderAttack condition is true in the model.[1] Conditions such as cLongAttack (line 3) and cFreqAttacker (line 9) reflect the state of architectural properties, such as whether an attack is ongoing, and act as guards in the strategy's decision tree. This strategy captures the scenarios where infrequent and frequent attackers are dealt with by throttling or blackholing the attack respectively, unless the attack is long running. This is consistent with the logic described above.

Listing 1: An example strategy for implementing the DoS Wrapper.

```
1  strategy FixDoSAttack [cUnderAttack] {
2    t0: (cInfreqAttacker) -> throttle() @[2000(/*ms*/ ]
         {
3      t0a: (cLongAttack) ->  blackhole () @[2000] {
4        t0ai: (default) -> done;
5      }
6      // No more steps to take
7      t0a: (default) -> TNULL;
8    }
9    t1: (cFreqAttacker) -> blackhole() @[2000/*ms*/] {
10     t1a: (cLongAttack) -> blackhole() @[2000] {
11       t1ai: (default) -> done;
12     }
13   }
14 }
```

While strategies determine which action to take, tactics are responsible for taking the action. If a condition is true in the strategy, then the subsequent tactic is applicable. For example, an infrequent attacker would cause the cInfreqAttacker condition to be true invoking the throttle() tactic (line 2). Tactics are the specific actions to take to transform the architecture into a desired state. The tactics used in the FixDoSAttack strategy are: throttle and blackhole.

Consider the blackhole tactic shown in Listing 2. When executed, this tactic will change the ZNN system to discard traffic from specified clients (i.e., put them in a blackhole).

[1]The details of this condition are elided for space, but are written in the first-order predicate langauge of Acme [10]

The tactic has three main parts: the applicability condition that determines whether the tactic is valid for the situation, the tactics action on the architecture, and the tactic's anticipated affect on the model of the system.

Listing 2: Tactic to black hole an attacker.

```
1  tactic blackholeAttacker () {
2   condition {
3     // check any malicious clients
4     cUnblackholedMaliciousClients
5   }
6   action {
7    // Collect all attackers
8    set evilClients =
9      { select c : T.ClientT in M.components |
10       c.maliciousness > M.MAX_MALICIOUS};
11   for (T.ClientT target : evilClients) {
12     // black hole the malicious attacker
13     Sys.blackhole(M.lbproxy, target);
14   }
15  }
16  effect {
17   // all the malicious clients blackholed.
18   !cUnblackholedMaliciousClients
19  }
20 }
```

The tactic's applicability condition relies on whether or not a client is currently attacking, indicated by a maliciousness threshold property in the architecture. If a gauge sets the maliciousness property of the suspected attacker above the maliciousness threshold, then blackholing is a viable action to take. In this sense, the tactic provides an additional checkpoint that more closely aligns with the architecture.

The action part of the tactic places clients that are identified as attackers in the blackhole by invoking the Sys.blackhole(...) operation. This operation is bound to an effector that actually modifies the Znn system to drop attacker packets, by adding the client to the blacklist, adding the filter, and restarting the load balancer.

Finally, the effect section verifies that the action had the intended impact on the system. In terms of acting as a protective wrapper, this tactic adapts Rainbow that it can intercept and discard malicious traffic before it reaches the Znn infrastructure.

7. CONCLUDING REMARKS

This paper has illustrated the benefits of evaluating the security properties of a software system using architectural models, and in particular at runtime. In support of our argument, we illustrated (1) how existing ad hoc techniques to self-protection can be formulated as architecture-level patterns, thus paving the way for a systematic engineering approach to construction of such systems, and (2) how those patterns could be realized in Rainbow. Our experiences with extending Rainbow to protect web applications have corroborated some of the hypothesized benefits of architecture-based self-protection (recall section 4). For instance, by explicitly separating the DoS detection and mitigation capability from the application logic, our solution can be easily reused in any other web application managed by Rainbow. In addition, by representing the self-protection capabilities as architectural elements (e.g., a protective wrapper connector), our solution allows us to reason about the security posture of a system in terms of its architectural configuration. This also enables adaptation of self-protection mechanism itself, e.g., addition and removal of new wrapper connectors.

While our experiences have been very positive so far, a number of research challenges remain:

Quantifying security. One of the difficulties in automatically making adaptation decisions is the lack of established and commonly accepted metrics for the quantification of security. Good security metrics are needed to enable comparison of candidate adaptations, evaluate the effect adaptations have on security properties, and quantify overall system security posture. However, there are few security metrics that can be applied at an architectural level.

Architectural-level metrics are preferred because they reflect the system security properties affected by adaptations. Such metrics could include measures to classify security based on applied adaptations and evaluate the impact of adaptations on certain security properties (e.g., attack surface, number least privilege violations). We have done some preliminary work in quantifying the attack surface with respect to an architecture in [11]. With the aid of such metrics, in our future work we plan to develop better mechanisms to better select adaptations that promote desired properties and quantify the impact on system security.

Quality attribute tradeoffs. In fielded systems security must be considered with other, possibly conflicting, quality attributes. Rainbow makes tradeoffs between quality attributes with each adaptation, selecting an adaptation that maximizes the overall utility of the system. Principled mechanisms are needed to evaluate the impacts of these tradeoffs as the system changes. Consider that almost all self-protection patterns described in Section 5 come at the expense of other quality attributes (e.g. response time, availability). Mechanisms to automatically evaluate competing quality attributes is critical for effective self adaptation.

Software architecture is the appropriate medium for evaluating such tradeoffs automatically because it provides a holistic view of the system. Rainbow reasons about a multi-dimensional utility function, where each dimension represents the user's preferences with respect to a particular quality attribute, to select an appropriate strategy. We plan to evaluate this approach with respect to security, which requires quantifying security as above.

Protecting the self-protection logic. Most of the research to date has assumed the self-protection logic itself is immune from security attacks. One of the reasons for this simplifying assumption is that prior techniques have not achieved a disciplined split between the protected subsystem and the protecting subsystem. In our approach, the inversion of dependency and clear separation of application logic from Rainbow present us with an opportunity to address this problem. The inversion of dependency allows us to layer the self-protection logic recursively, and thus have one instance of Rainbow protect another instance of it. The fact that the two subsystems are separated allows us to leverage techniques such as virtualization, such that Rainbow would execute on a separate virtual machine, thereby reducing the likelihood of it being compromised by the same threats targeted at the application logic.

As we explore the avenues of future research outlined above, we hope the paper provides an impetus for others to do the same.

8. ACKNOWLEDGMENTS

This work is supported in part by awards W911NF-09-1-0273 from Army Research Office, D11AP00282 from Defense

Advanced Research Projects Agency, and CCF-1217503 and CCF-1252644 from National Science Foundation.

9. REFERENCES

[1] Barna, C. et al. Model-based adaptive dos attack mitigation. In *International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)* (2012), pp. 119–128.

[2] Casanova, P. et al. Diagnosing architectural run-time failures. To appear in SEAMS, 2013.

[3] Casanova, P. et al. Architecture-based run-time fault diagnosis. In *Proceedings of the 5th European Conference on Software Architecture* (2011).

[4] Castro, M., and Liskov, B. Practical byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst. 20*, 4 (Nov. 2002), 398–461.

[5] Cheng, S.-W., and Garlan, D. Stitch: A language for architecture-based self-adaptation. *Journal of Systems and Software, Special Issue on State of the Art in Self-Adaptive Systems 85*, 12 (December 2012).

[6] Cheng, S.-W. et al. Evaluating the effectiveness of the rainbow self-adaptive system. In *ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems, 2009. SEAMS '09* (May 2009), pp. 132 –141.

[7] Chess, D. M. et al. Security in an autonomic computing environment. *IBM Systems Journal 42*, 1 (2003), 107–118.

[8] Foo, B. et al. ADEPTS: adaptive intrusion response using attack graphs in an e-commerce environment. In *International Conference on Dependable Systems and Networks, 2005. DSN 2005. Proceedings* (July 2005), pp. 508–517.

[9] Garlan, D. et al. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *IEEE Computer 37*, 10 (Oct. 2004), 46–54.

[10] Garlan, D. et al. Acme: Architectural Description of Component-Based Systems. In *Foundations of Component-Based Systems*, G. Leavens and M. Sitaraman, Eds. Cambridge University Press, 2000, pp. 47–68.

[11] Gennari, J., and Garlan, D. Measuring attack surface in software architecture. Tech. Rep. CMU-ISR-11-121, Institute for Software Research, School of Computer Science, Carnegie Mellon University, 2011.

[12] Hafiz, M. et al. Organizing security patterns. *IEEE Software 24*, 4 (Aug. 2007), 52–60.

[13] Huang, Y. et al. Software rejuvenation: analysis, module and applications. In *, Twenty-Fifth International Symposium on Fault-Tolerant Computing, 1995. FTCS-25. Digest of Papers* (June 1995), pp. 381–390.

[14] Kephart, J., and Chess, D. The vision of autonomic computing. *Computer 36*, 1 (Jan. 2003), 41–50.

[15] Kitchenham, B. Procedures for performing systematic reviews. *Keele, UK, Keele University 33* (2004).

[16] Li, M., and Li, M. An adaptive approach for defending against ddos attacks. *Mathematical Problems in Engineering* (2010).

[17] Nagarajan, A. et al. Combining intrusion detection and recovery for enhancing system dependability. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops (DSN-W)* (June 2011), pp. 25 –30.

[18] Nguyen, Q., and Sood, A. A comparison of intrusion-tolerant system architectures. *IEEE Security Privacy 9*, 4 (Aug. 2011), 24–31.

[19] North, D. A tutorial introduction to decision theory. *IEEE Transactions on Systems Science and Cybernetics 4*, 3 (1968), 200–210.

[20] Okhravi, H. et al. Creating a cyber moving target for critical infrastructure applications using platform diversity. *International Journal of Critical Infrastructure Protection 5*, 1 (Mar. 2012), 30–39.

[21] OWASP.org. Cross-site scripting (XSS) - OWASP. https://www.owasp.org/index.php/Cross-site_Scripting_(XSS).

[22] OWASP.org. Owasp top ten project. https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project.

[23] Sibai, F., and Menasce, D. Defeating the insider threat via autonomic network capabilities. In *2011 Third International Conference on Communication Systems and Networks (COMSNETS)* (Jan. 2011).

[24] Sousa, P. et al. Highly available intrusion-tolerant services with proactive-reactive recovery. *IEEE Transactions on Parallel and Distributed Systems 21*, 4 (Apr. 2010), 452–465.

[25] Stakhanova, N. et al. A taxonomy of intrusion response systems. *International Journal of Information and Computer Security 1*, 1 (Jan. 2007), 169–184.

[26] The MITRE Corporation. CWE - 2011 CWE/SANS top 25 most dangerous software errors. http://cwe.mitre.org/top25/.

[27] The MITRE Corporation. CWE-89: improper neutralization of special elements used in an SQL command ('SQL injection'). http://cwe.mitre.org/data/definitions/89.html.

[28] Valdes, A. et al. An architecture for an adaptive intrusion-tolerant server. In *Security Protocols*, B. Christianson, B. Crispo, J. Malcolm, and M. Roe, Eds., vol. 2845 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2004, pp. 569–574.

[29] Wang, F. et al. SITAR: a scalable intrusion-tolerant architecture for distributed services. In *Foundations of Intrusion Tolerant Systems, 2003* (2003), pp. 359–367.

[30] Yoshioka, N. et al. A survey on security patterns. *Progress in Informatics 5*, 5 (2008), 35–47.

[31] Yuan, E., and Malek, S. A taxonomy and survey of self-protecting software systems. In *International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)* (2012).

[32] Yuan, E. et al. A survey of self-protecting software systems. In *ACM Transactions on Autonomous and Adaptive Systems (TAAS)* (June 2013).

[33] Zhu, M. et al. VASP: virtualization assisted security monitor for cross-platform protection. In *Proceedings of the 2011 ACM Symposium on Applied Computing* (2011), pp. 554–559.