Selfware : Lessons Learned to Build Autonomic Systems

Abstract— **Autonomic computing[14] aims at enabling computing infrastructures to perform administration tasks without (or with minimal) human intervention. This paper reports the experience we gained with the design and the experiments of Selfware, an architecture-based autonomic system. The contribution of this paper is (i) to advocate the use of a reflexive component-based approach for the design of autonomic architecture-based management systems, (ii) to present the design of three autonomic control loop for self-protection, self-healing, and self-optimization, and (iii) to report our experience and lessons learned in their use for autonomic management of web servers and message-oriented middleware in cluster environments.**

I. Introduction

The goal of autonomic computing [14] is to automate the functions related to system administration. This effort is motivated by the increasing size and complexity of systems and applications alike, which has two direct consequences: the administration costs are an increasing part of the total information system costs, the difficulty of the administration tasks reach the limits of what human administrators can handle. Consequently, autonomic computing advocates self-management capabilities.

In SELFWARE, we approach self-management through an architecture-based approach where self-management is about observing and evolving the architecture of the managed systems. The role of autonomic managers is to react to observed evolutions of the managed system, re-architecturing it accordingly. Our experience is that a reflexive component-oriented approach is very effective for self-management capabilities. SELFWARE uses components to capture the traditional concept of managed elements but applies component modeling to not only managed applications but also the architecture and behavior of the underlying distributed system hosting these applications. Legacy systems are wrapped as components to provide reflexive capabilities over the legacy configurations and a uniform management interface.

In other words, SELFWARE models and controls the architecture and behavior of a complete distributed system through a component-oriented approach.

In particular, SELFWARE also advocates the use of component reflection as the foundation to support the *introspection* and *reconfiguration* of the managed distributed system. Through instrospection, autonomic managers can not only observe the architecture of the managed distributed system but also its runtime behavior, including for instance its dynamic performance for detecting poor resource utilization or security-related communication patterns for detecting intrusion. Through reconfiguration, autonomic managers could manipulate the architecture of the managed distributed system, including for instance the ability to provide higher availability through replicating components across nodes, or changing the overal Quality of Service (QoS) by replacing certain components with others that have different QoS characteristics.

SELFWARE advocates a minimal reflection on components that offers a small but powerfull management interface that capture traditional management aspects such as configuration properties, bindings amongst components and lifecycle.

We report here about our experience wrapping two legacy distributed systems: a clustered multi-tiered web server and a message-oriented middleware using a snowflake distributed architecture. Wrappers have been small and easy to write, only wrapping low-level mechanisms, thereby easily enabling autonomic managers to control these systems. We have experimented three managers, providing self-protection, self-optimization and self-repair capabilities. It is our experience that the development of such managers is greatly simplified by not only an architecture-based approach but also by our reflexive component-oriented design.

This paper is organized as follows. In Section II, we present Selfware design. In Section III, IV and V, we present respectively our self-protection, self-optimisation and self-repair managers. Section VI gives feedbacks about our wrapping experiments and our managers. Section VII present the related works. In Section VIII, we conclude.

II. Selfware Design

The design of SELFWARE follows an architecture-based approach which enable a good level of gener-

icity. Indeed autonomic managers work on system's structure and configuration which are common concepts to all legacy distributed systems. Thus a manager observes and control the hardware and software architecture of the distributed system it manages. In other words, it needs to know about the managed subsystems and the machines hosting them. It also need to know the communication channels between subsystems. This is achieved through the use of components, in several steps.

First, legacy subsystems are wrapped by thin components, called *wrappers*, that provide the ability to remotely manage them. For instance, wrappers allow to start-stop legacy systems individually as well as to configure the communication channels they use.

Second, we designed SELFWARE itself using the very same component model that where used for writing wrappers, opening up the possibility of SELFWARE managing SELFWARE. Indeed, SELFWARE does not only model and manage a distributed system, it is a distributed system composed of both *wrappers* and *Selfware components* which represent the internals of SELFWARE. The architecture of both the managed system and SELFWARE internals are represented in a *managed architecture* that captures consistent architecture state and provides not only a full description of the overall distributed system but also the ability to reconfigure it entirely. Using this managed architecture, autonomic managers of SELFWARE introspect the managed system, detect abnormal situations, and correct them through reconfiguring the architecture.

A. Wrapping legacy systems

Wrapping legacy systems is the first step towards self-management, bridging from the heterogeneous world of legacy systems to the homogeneous world of components. Each managed legacy system is wrapped as one FRACTAL component [5]. Wrappers offer a small but powerfull set of management operations that empower SELFWARE to achieve many autonomic behaviors such as self-protection, self-repair or self-optimization.

Each wrapper supports both the introspection and reconfiguration of the configuration it reifies. In particular, provides control over the lifecycle, the attributes, and bindings of the legacy subsystem it wraps. The lifecycle is about starting and stopping a wrapped legacy system. The attributes capture, as key-value pairs, the configuration data of the wrapped legacy system, usually found in configuration files. In other words, legacy systems are configured through setting attribute values. Attribute values may be any value that can be copied and serialized to an external format. Attribute values may be also names to outside resources such as file names, database names, or URLs.

Bindings capture the presence of communication channels between legacy systems. It is important to point out that bindings capture the existence of communication channels but they are not involved in the actual communication. Wrapped legacy systems continue to communicate directly as they would have without being wrapped. Simply put, wrapper components and bindings actually show the overall assembly of legacy systems, as depicted in the bottom two layers of the Figure 1.

We decided to use Java, exploiting the distributed Java incarnation of FRACTAL. Indeed, wrappers are co-located with their legacy systems since most management operations are better achieved locally, but this means that wrappers are distributed across machines as managed legacy systems are. Therefore, as remote Java objects, wrappers provide a simple way for SELFWARE to remotely manage legacy systems.

It is important to point out that wrappers must adhere to a fail-stop semantic. Such semantic requires not only wrappers to be fail-stop but also that failing wrappers actually stop their wrapped legacy system before they fail. Thus either a wrapper fully applyies the new configuration to the wrapped legacy or both are fail-stop. This semantic of the wrappers ensures that failures during repair attempts do not compromise the consistency between the configuration of the legacy system and its reification at wrapper level. It is our experience that such fail-stop assumption is realistic and an important requirement to build self-* properties.

Despite distribution, wrappers are simple Java components that provides management operations through a few set of Java interface. The implementations of the corresponding methods is often reduced to manipulating configuration files or launching already existing scripts. For instance, wrapping the Apache HTTP daemon relies on the start and stop scripts for implementing the lifecycle operations.

B. Recursive design

A SELFWARE management system consists of *wrappers*, *autonomic managers* and the *managed architecture*.

The managed architecture (depicted in Figure 1) provides the full description of the hardware and software architecture of the managed system. In the bottom two layers, the figure shows the managed distributed system, including hardware nodes, wrappers, and bindings between them. At the top, the managed

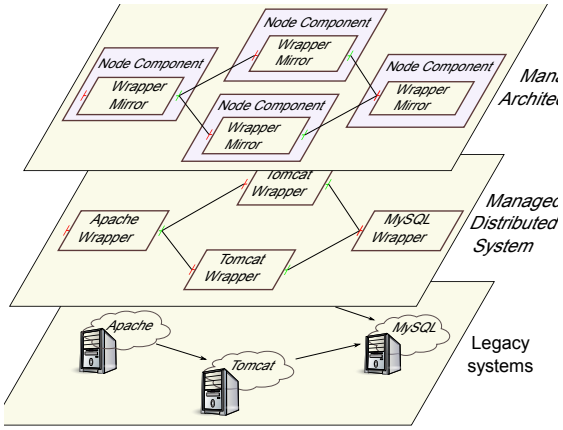

Fig. 1: Managed Architecture figure

architecture captures that complex architecture, using components. Hardware nodes are represented by *node mirrors*. Besides capturing the description and state of managed hardware nodes, node mirrors capture the knowledge of the components installed on their hardware node.

Wrappers are also represented by components, called *wrapper mirrors*. The wrapper mirrors are part of the managed architecture, entirely created and managed by SELFWARE . The mirrors capture the complete architectural state of their corresponding wrappers, hence their name. In other words, for each wrappers co-located with a legacy, one finds a corresponding wrapper mirror in the architecture. The architectural state captured by mirrors is directly related to the management operations provided by wrappers. It includes the knowledge of the lifecycle status (started or stopped). It also includes the knowledge of the attributes (key-value pairs). Finally, it includes the knowledge of the current bindings between wrappers.

Through mirrors, autonomic managers can both introspect and reconfigure the architecture. By introspecting, we mean that managers can access the mirrors and therefore introspect the architectural state they mirror. For instance, managers can know which wrappers are deployed where, if they are started, and what are the bindings that link them. By reconfigure, we mean that autonomic managers can change any aspect of the architecture they introspect. For instance, managers can start-stop wrappers or change attributes. They can also remove or create bindings between wrappers. This is simply done through manipulating the mirrors, using the management operations such as *start()*/*stop()*, *bind()*/*unbind()*, and *setAttribute()*. Such reconfigurations are done on the managed architecture, manipulating components. Internally and autonomously, SELFWARE will apply the reconfiguration onto wrappers, themselves applying the management operations on the legacy systems they wrap. In particular, a manager reconfigures the managed architecture in an atomic session that it commits; the actual commit of the session is the responsability of the SELFWARE runtime. The commit must carry over the reconfiguration from the managed architecture out to the concerned remote wrappers. It is important to point out that, when commiting, the reconfiguration has already been successful on the managed architecture.

Through node mirrors, managers can also control the deployment of wrappers. Like any other component, a node mirror may have bindings to other components. For node mirrors, SELFWARE introduces a deployment binding that carries the semantics of deployment. Hence, to deploy a wrapper on a hardware node, one only needs to bind a node mirror to the corresponding wrapper mirror. This binding tells SELFWARE to deploy the wrapper on the node and to maintain the association between the deployed wrapper and the wrapper mirror representing it in the architecture. SELFWARE uses OSGI as its underlying infrastructure to deploy Java components. For legacy systems, SELFWARE provide some support and may be able to deploy them as well, either through built-in mechanisms or through standard installers. However, in complex settings, manual installation is most often required.

Our design suggests that SELFWARE itself is built using the very same component model as the one used to wrap legacy systems. In other words, the internals of SELFWARE appear to SELFWARE as any other managed distributed system: composed of components providing the necessary management operations: lifecycle, attributes, and bindings. The goal is to use uniform Architecture-based concepts and API, based on introspecting and reconfiguring the managed architecture. In other words, we want to be able to manage identically either the managed distributed system or the SELFWARE runtime itself.

Although they use the same component model, it is important to distinguish between wrappers and SELFWARE components. Wrappers are Java components, co-located with the legacy systems they wrap, but wrappers are solely about providing remote management operations on legacy systems. While SELFWARE components are also Java components, they are no longer wrappers of legacy systems but they are fully

functional components used to implement the internals of SELFWARE . Although the goals and roles of wrappers and SELFWARE components are different, SELFWARE components and wrappers provide the very same management operations, which enables a uniform management.

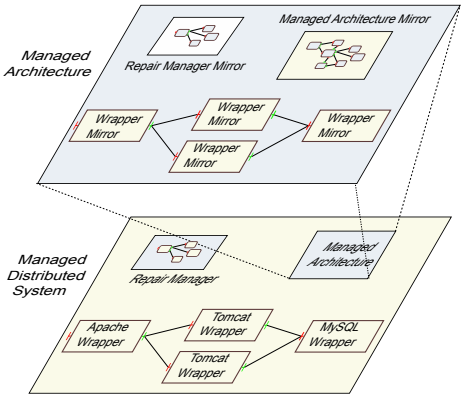

Fig. 2: Recursive Architecture figure

In particular, SELFWARE components are mirrored in the managed architecture, as depicted in Figure 2. The recursive nature of our design comes from that very fact: the managed architecture mirrors itself. The lower plane shows the managed system that includes not only the wrappers as before but also the managers (e.g. the repair manager) and the managed architecture. The top plane represents the mirrors in the managed architecture. We find the usual mirrors for the wrappers. We also find mirrors for the components implementing the managers and the managed architecture itself. The recursion stops as we do not mirror mirrors, but every component existing in SELFWARE has a mirror in the architecture, including the ones used to implemented the managed architecture. With this complete managed architecture, the same repair algorithm can detect and repair failures of the SELFWARE runtime as this will be explained latter.

III. SELF-PROTECTION

In the mid 90s, self-protection was approach as a *computer immune system*, inspired by natural immune systems that protect beings from foreign pathogens. An immune system relies on a *sense of self*, that is, the ability to detect the intrusion of foreign elements through the distinction of *self* from *non-self*. Once an intruder is detected, counter measures can be put in place to destroy it or at least contain progression and the damages it creates.

Our focus is on the needed low-level mechanisms for a global autonomic self-protection in a cluster environment and not on the development of new specific techniques for access control, intrusion detection, or backtracking. In particular, we focus on generic mechanisms that not only can recognize known and unknown attacks but also are independent from any specifics of wrapped legacy systems. Furthermore, our mechanisms must ensure zero false positives as counter measures are triggered autonomously.

We report here our work on detecting illegal communication channels. Our approach uses the knowledge of the application architecture as the decision knowledge for self-protection, detecting illegal communication patterns and isolating compromised components. This manager shows the interest of an architecture-based approach for self-protection and handles well the recursive challenge of self-protection: the immune system protecting a computer system may itself become the target of attacks. Since our immune system is a SELFWARE manager, designed using components, it can self protect through the same approach.

A. Self-knowledge

In SELFWARE , *self-knowledge* is defined in a legacy independent way since it is entirely architecture-based. Focusing on illegal communication channels, we need to have the knowledge of legal components and legal communication channels between these components. The *Managed Architecture* captures the necessary information. Remember that Node Mirrors allow to discover which softwares are running on cluster's nodes and how these softwares are interconnected. The bindings reify the TCP/IP parameters involved by the communications. It is easy to infer from the *managed architecture* which communication channels are legal between each nodes.

Indeed, any communication through a network connection that does not corresponds to an existing binding between known components in the architecture is considered an attack and must be blocked. This approach has no false positives if we assume an accurate architecture, which SELFWARE provides if we assume that only legal architecture manipulations are allowed. To ensure this, all reconfiguration requests are authenticated through asymmetric cryptography, making sure only official autonomic managers are allowed to reconfigure the architecture of the managed system through the *managed architecture*. This prevents a compromised component to manipulate the architecture and breaking the invariant of our approach. From

this knowledge, attacks can be detected, as explained below.

B. Protection mechanism

Our protection mechanism relies on managed firewalls, one such firewall running on each node of the cluster. In our prototype, we used the *netfilter* firewall that we wrapped so that it can be managed like any other SELFWARE component. Firewalls are configured automatically from the self-knowledge available in *managed architecture*. Everytime this architecture evolves, firewall configurations are updated accordingly. This is done by the self-protection manager that watch the Selfware architecture reconfiguration and maintain the firewall configurations in sync using the binding knowledge of the communication ports and IP addresses of established communication channels between managed components on the cluster.

When detected, illegal communications between nodes are prevented by firewalls and the self-protection manager is notified. Different policies may be implemented; in our prototype, we chose to consider the node from which the illegal communication originates as compromised. The rationale is that only a compromised node in a SELFWARE -managed cluster can attempt an illegal communication. The self-protection manager will therefore reconfigure the architecture in order to isolate the compromised node.

Figure 3 depicts such isolation of a compromised node in the context of a multi-tiered application server. The firewall on node 4 detects an illegal communication originating from node 5. It prevents it and notifies the self-protection manager that reconfigures the architecture in order to isolate node 5.

IV. SELF-OPTIMIZATION

In this section, we discuss the means of implementing self-optimization in replicated cluster-based systems. Optimization is defined using two main criteria: performance, as perceived by the clients (e.g. response time) or by the application's provider (e.g. global throughput); and resource usage (e.g. processor occupation). One may then define an "optimal" region using a combination of these criteria. Providing self-optimization for a system consists in maintaining the system in the optimal region, in the presence of dynamic changes, e.g. widely varying workload. Here, we consider implementing self-optimization using resizing techniques, i.e. dynamically increasing or decreasing the number of nodes allocated to the application.

A classical pattern for implementing scalable clustered servers is the load balancer. In this pattern, the server is statically replicated on start-up and a load balancer is placed in front of the servers. The load balancer aims at sharing the load between each servers. A request can be handled indifferently by one of the server. On a new request, one server run the required service, ensures the consistency of its state with the others servers and return the response to the client.

Selfware aims at autonomously increasing/decreasing the number of replicated servers when the load varies. This has the effect of efficiently maximizing servers utilization (i.e. no resource overbooking) to ensure end user needs with no human intervention required.

All start by a manager which detects the overload or the underload of the clustered server and then reconfigures the wrapper mirrors corresponding to the legacy server replicas and the load-balancer. In particular, the control loop has the following sensors and actuators.

The *sensor* periodically measure the chosen performance (or QoS) criteria, i.e. a combination of CPU usage and user-perceived response time. A probe is therefore deployed on each node to know the load of each server. Each probe computes a moving average of the collected data in order to remove artifacts characterizing the CPU consumption. The sensor finally computes an average load across all involved nodes from probe's reports, so as to observe a general load indication of the whole replicated server.

Thanks to the uniform management interface provided by Selfware, the actuators are generic, since increasing or decreasing the number of replicas which belong to the replicated server is implemented as adding or removing wrapper mirrors in the *managed architecture* and binding/unbinding them from their load-balancer.

In our experiments, the decision logic is based on thresholds on loads provided by sensors. When more replicas are required, the manager allocates an empty node mirror, adds a new server mirror (The required wrapper is therefore created on the corresponding node) and integrates the new server with the load balancer. Similarly, if the replicated server is underutilized, the main operations performed by the manager on the *managed architecture* are to stop these replicas, unbind them from the load balancer, and release the nodes hosting these replicas if no longer used.

It is important to point out that replicated server may contain a state which must be consistent between each replicas. Applying a load balancing policy without a mechanism to synchronize the state of the

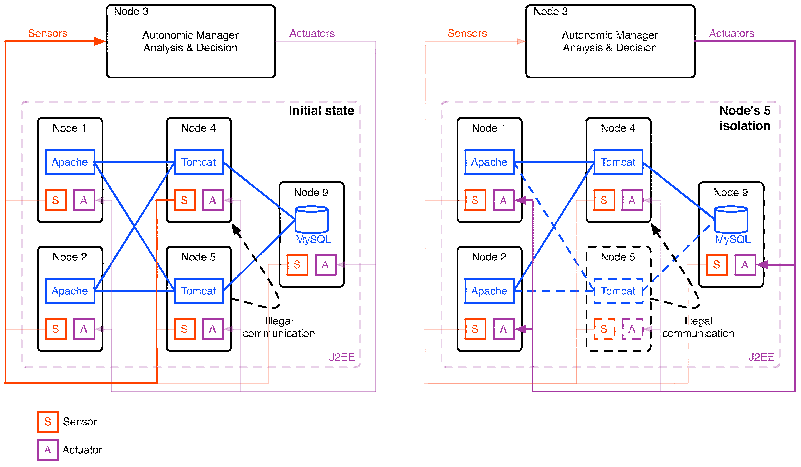

Fig. 3: Attack detection and isolation of node 5

figure

servers may lead to inconsistencies. In the case of self-optimization, if the server part does not contain a state, the sizing of the replicated server does not induce any problem. In the case where the replicated server maintains a state, it is the responsability of the wrappers to drive the state synchronization between replicas. In our experiments, this was very simple because the legacy servers provide mechanisms to synchronize the replicated servers between each other. It is our experience that modern legacy servers that conform to the loadbalancing pattern described above provide such mechanisms. In particular this is the case of replicated servlet containers, replicated JMS servers and replicated database servers, where load balancers (e.g. C-JDBC) play also the role of replication consistency manager [7].

The self-optimization manager we have described here can be used with any legacy server that conforms to the given loadbalancing pattern. Indeed we reuse the very same manager to control a cluster of database and a cluster of tomcat servers. Whereas we do not have experimented self-self-optimization on *Selfware* itself, we could reuse the same manager for the self-self optimization of a cluster of *Selfware*. This latter case requires *Selfware* internals to be replicated following the same loadbalancing pattern. Instead we choose to illustrate the self-self capabilities in the case of the repair algorithm presented in the next section.

V. Self-repair

Self-repair is certainly one of our most advanced autonomic manager in SELFWARE . It shows the interest of our architectural concepts and recursive design to provide self-self-repair. In other word, it ensures not only the autonomic self-repair behavior of managed applications but also it ensures the same self-repair property for itself using the very same algorithm.

A. Architecture-based self-repair

Our self-repair advocates an architectural recovery process that, after a failure, re-establishes a valid software architecture of the managed system.

Our current failure model is fail-stop model for nodes. When one or more failures are detected, the self-repair manager analyzes the failures by introspecting the managed architecture and repairs these failures by reconfiguring the architecture.

Remember that introspection and reconfiguration happen in one session on the managed architecture only capturing consistent architectural state atomically. At any time up to the final commit of the session, the self-repair manager can abort the session.

During the analyzis step, the repair manager identifies the impacts of the detected failures, determining the set of *failed components* that were lost due to the node failures. Indeed, a single node may host many components. This requires introspecting the managed architecture in order to discover all the components that were deployed on the nodes that failed.

Using the managed architecture, the self-repair manager can not only know which wrappers have been lost to the detected failures but also their complete architectural state.

The repair policy ensured by our repair manager is to substitute failed nodes by new ones from a pool of available hardware nodes, reconfiguring these

new nodes exactly as the lost ones were configured. This includes not only deploying wrappers and their wrapped legacy systems but also reconfiguring wrappers once deployed.

One important point is that the new node must be chosen in a way that allows re-deployed legacy systems to still access their persistent states. The specifics here are highly dependent on the kind of wrapped legacy systems but remote database connections or distributed file systems are possible examples of mechanisms used to achieve this. In the architecture, the necessary knowledge is captured through compatibility tables between hardware nodes and wrapped legacy systems. Choosing a node where to recreate a component is therefore as simple as choosing a compatible and available node in the compatibility tables. Creating compatible nodes is pre-required work from human administrators.

Recreating and reconfiguring lost wrappers is only one half of the repair reconfiguration; the other half is about cleaning up stale bindings before creating correct ones. This requires to compute the set of *impacted components*. These impacted components are all the components currently bound to a failed component. The knowledge of impacted components enables the self-repair manager to cleanup stale bindings leading to failed components.

This cleaning up of stale bindings is simply done by unbinding them in the managed architecture. When applying these unbind operations, SELFWARE will forward the unbind operations to the wrappers of impacted components and they will request their wrapped legacy to close stale communication channels. The creation of new bindings is also simply done on the managed architecture. When applying these bind operations, SELFWARE will forward these bind operations to the wrappers of impacted components, allowing wrappers to inform their legacy of the new communication channels to use.

B. Recursive self-repair

Self-repair suggests a recursive design that exploits the overall recursive design of SELFWARE , based on components. The self-repair manager watches over managed components and repairs them when a failure is detected; but the self-repair manager itself needs to be watched and repaired in case a failure affects its operation. We term this the self-self-repair.

For self-self-repair, we need to add fault-tolerance to both the self-repair process itself and the managed architecture that is used by the self-repair process.

Indeed, SELFWARE can provide no continuous self-repair and certainly no self-self-repair if SELFWARE fails. A recursive design alone does not prevent SELFWARE from failing, it opens the possibility to repair a failed SELFWARE runtime using the repair technics implemented by SELFWARE . This suggests our solution: replicate SELFWARE to let valid replicas of SELFWARE repair the failed replicas of SELFWARE . The repair of SELFWARE is made easier because both the managed architecture and the self-repair manager are both developed as SELFWARE components.

Therefore using component-level replication, we can replicate all SELFWARE components as depicted in Figure 4.

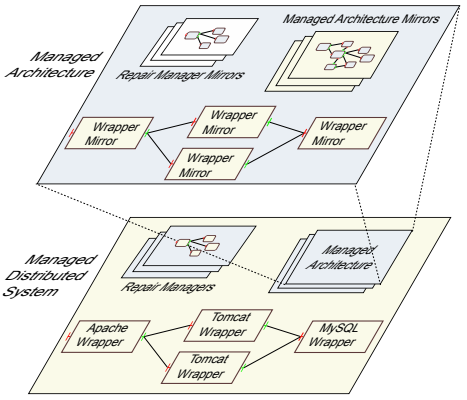

Fig. 4: Replicating SELFWARE figure

It is important to point out that replication only concerns SELFWARE components; as aforementioned, these SELFWARE components are the internals of the managed architecture and autonomic managers. Therefore, we neither replicate legacy systems nor their wrappers. SELFWARE components are deterministic and can therefore be transparently replicated using an active replication scheme.

By being a replicated component, the self-repair manager can self-repair itself. First, the self-repair manager detects failures of replicas of itself. Second, it is able to re-create failed replicas, like any other component lost due to a failure. In doing so, the self-repair manager maintains the cardinality of its own replication, without human intervention.

Remember that the semantics of the wrappers ensures that failures during repair attempts do not compromise the consistency between the configuration of the legacy system and its reification at wrapper level. Being a replicated component, the managed architecture is also guaranteed to resist node failures. We also know that the managed architecture only captures consistent architecture state since it is updated atomically.

The operating principle to gain self-self-repair capability can be summarized as follows. First, there is one self-repair manager that uses one managed architecture. Second, both are implemented as SELFWARE components that are transparently replicated by SELFWARE . Replication provides a first step towards self-self-repair; the second step is provided through the ability of the self-repair manager to maintain the replication cardinality. Third and final step, the consistency of the whole architecture is ensured by the atomicity of architecture reconfigurations.

VI. EVALUATION

We report here our experience in wrapping a multi-tier application server and a Message-Oriented Middleware then we evaluate the intrusivity induced by the Selfware management system as well as the efficiency of the self-optimization and the self-repair managers. These managers has been evaluated with the Rubis application benchmark which implements an auction site [1]. We used the Rubis 1.4.2 version of the multi-tier J2EE application running on several middleware platforms: Apache 1.3.29 as a web server [19], Jakarta Tomcat 3.3.2 as an enterprise server [23], MySQL 4.0.17 as a database server [15], Tomcat clustering as the enterprise server clustering solution [23], and c-jdbc 2.0.2 as the database server clustering system [7]. Experiments were performed on the Linux kernel running x86-compatible machines, with 1GB RAM and 1800MHz, connected via a 100Mb/s Ethernet LAN to form a cluster.

A. Wrapping

we give below the details of two real wrapping experiences: a J2EE clustered Web Application Server and an advanced JMS provider using a snowflake distributed design.

1) Multi-tier Application Server : A multi-tiered architecture is classically divided in three tiers: the HTTP daemon (Apache), the application server (Tomcat), and the database tier (MySQL for e.g.). This multi-tiers infratructure is managed through a set of interconnected components that wrap the tiered legacy system. Each wrapper provides the management operations defined by SELFWARE .

Attribute management is used to expose and change configuration attributes of the different tiers. For the web tier, it wraps the configuration file of the Apache HTTPD server. Hence, a modification of the port attribute of the Apache component is reflected in the *httpd.conf* file in which the port attribute is defined.

Binding management is used to reflect and manipulate connections between tiers. For instance, the Apache HTTPD server needs to be connected to the Tomcat Servlet engine. The implementation of this bind method is reflected at the legacy layer in the *worker.properties* file used to configure the connections between Apache and Tomcat servers. A bind operation will create that connection where an unbind operation removes that connection.

Lifecycle management are used to start and stop tiers. For instance, the web and presentation tiers can be started or stopped through the execution of shell scripts to start/stop the Apache HTTPD server or Tomcat Servlet engine.

To achieve higher availability, each tier can be replicated using the load balancing pattern presented section IV. In this case each load balancer is also wrapped as a component with bindings that capture its connections to the different servers it balance.

2) Message-Oriented Middleware: Message-Oriented Middlewares (MOMs) are distributed platforms that enable a message-based integration of loosely-coupled heterogeneous distributed systems. We wrapped the MOM called JORAM (Java Open Reliable Asynchronous Messaging) that provides a fully compliant implementation of the JMS specification. JMS applications cooperate through messages, using either message queues for point-to-point communications or topics for a publish-subscribe paradigm. In this context, architecture-based management is two-fold. On the one hand, one needs to manage message queues and topics. On the other hand, one needs to manage the distributed snowflake architecture of the JORAM middleware itself. The JORAM snowflake architecture is a distributed middleware that sets up a routing overlay for delivering queue and topic messages efficiently and reliably. In SELFWARE , JORAM servers, topics and queues are wrapped as components. Modeling message queues and topics as components allows for JMS administrative tasks to be done through Selfware. For example, one can create message queues or topics on certain JORAM servers. Again attributes reify the configuration of a JORAM server, a Queue or a Topic. Bindings between servers reify the routing overlay set up by the servers while a binding between a server and a queue (or a topic) reifyies the fact that the queue (resp. the topic) is locally created on the server.

JORAM supports replicated topics that can easily be modeled and managed through SELFWARE like we already discussed for replicated multi-tiered web servers. The same would be true of the JORAM advanced support for message queues with load-balancing capabilities. Going further, Selfware opens

the path for more autonomic management functions such as autonomic load balancing for both message queues and replicated topics.

The table I presents the code size of wrappers we talked about for the legacy systems and their ADL[1] description. Even for quite complex legacy systems, most wrappers where extremely simple and very similars to regular administration scripts.

			# Java lines	# ADL lines
Specific code	jee	Rubis app. - Web	150	11
		Rubis app. - Servlets	150	11
		Rubis app. - Database	150	11
		Total	***450***	***33***
		Apache Web server	800	16
		Tomcat Servlet container	550	12
		MySQL SGBD	760	40
		Total	***2110***	***68***
	jms	joram server	368	51
		jndi	134	12
		jms Queue	253	16
		jms topic	297	16
		Total	***1052***	***95***

TABLE I: Wrapper code table

Regarding these experiments, we feel confident that a reflexive component-oriented approach to architecture-based management provides a good level of abstraction to wrap different kind of legacy systems.

B. Intrusivity

We evaluated the intrusivity induced by the Selfware management system. The intrusivity has been measured by comparing two executions of the Rubis application: when it is run and managed by Selfware and when it is run by hand, without Selfware. During this evaluation, the J2EE application has been submitted to a medium workload so that its execution under the control of Selfware didn't induce any dynamic reconfiguration. This intrusivity is quantified in terms of average throughput, response time and memory usage of the J2EE application in Table II.

During our experiments, we simulated about 300 web clients generating a medium workload. Without failures, any difference in performance would be representative of the overhead of SELFWARE. As the following table shows, our measures show no significant overhead in terms of application response times and throughput.

We can notice a slight memory overhead (20.1% vs. 17.5%) that can be linked with the creation of internal software components by SELFWARE. However,

[1]Architecture Description Langage (ADL) is the declarative language used to describe component's bindings and attributes

	without Selfware.	with Selfware.
Throughput	12 req./s	12 req./s
Resp. time	87 ms	89 ms
Mem. usage	17.5 %	20.1 %

TABLE II: Rubis experiments, with and without Selfware.

table

SELFWARE does not induce a perceptible overhead on throughput; this is due to the fact that SELFWARE does not intercept application communications but only configuration/management operations.

C. Self-optimization

In order to evaluate the self-optimization policy provided by Selfware, we considered a scenario where the application workload varies dynamically. At the beginning of the experiment, the web application is submitted to a medium workload (80 clients); then the load increases progressively up to 500 clients; and finally the load decreases symmetrically down to reach 80 clients.

Initially, the multi-tier auction site is deployed on one enterprise (and web) front-end server and one database back-end server. Since CPU is the only bottleneck resource in these experiments, the managed system elements (i.e. enterprise tier and database tier) are monitored by sensors that gather CPU usage information every second and compute a spatial [2] and temporal [3] average CPU usage value [4]. The self-optimization manager ensures that the average CPU usage is kept between a minimum and a maximum threshold. In order to prevent oscillations due to parallel reconfigurations started on the front-end tier and the back-end tier of the multi-tier managed system, the manager associated with the underlying Muti-Tier Managed Element specifies that a reconfiguration started on one of the tiers inhibits any new reconfiguration for a short period.

Figure 5 shows the variation of the number of replicas, for both the enterprise servers and database servers when the application workload varies. As the workload progressively increases, the average resource consumption of the cluster of replicated database systems also increases, and this tier becomes a bottleneck. An allocation of a new database replica is triggered, which results in a clustered back-end containing two

[2]Over nodes of replicated elements.

[3]Over the last 60 seconds.

[4]However, if CPU is not the only bottleneck resource, more sophisticated sensors consisting of an aggregation of single resource sensors might be used.

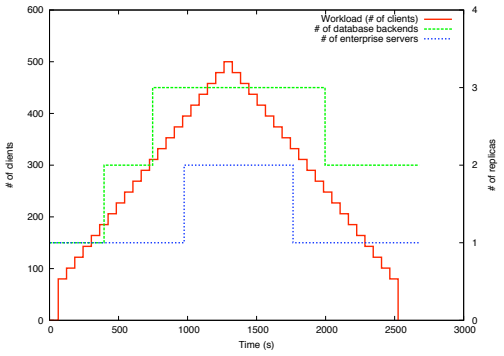

Fig. 5: Variation of the application workload and number of replicas
figure

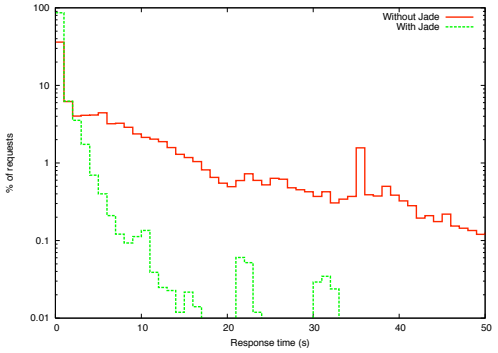

Fig. 6: Web client response time variation
figure

database systems. The workload continues growing and triggers another node allocation for the clustered database. The workload increases further; and this places the bottleneck on the front-end tier. An allocation of a new enterprise is triggered, resulting in a system composed of two enterprise systems and three database systems. The workload then increases without saturating this configuration before it starts decreasing. This workload decrease implies a decrease of the resource consumption of the front-end tier which triggers a de-allocation of one of its replicas, and then a low resource consumption of the clustered database, which triggers a de-allocation of a database replica. Figure 6 shows the response time variation as perceived by client with SELFWARE and without SELFWARE . The overall point here is that SELFWARE improves the overall response time by managing dynamically the number of required replicas. Also this illustrates well the interest of self-optimization based on architectural knowledge and reconfiguration to optimize various replicated legacy servers (e.g. a cluster of enterprise servers and a cluster of database in this experiment).

D. *Self-repair*

In this scenario, we consider the autonomic management of a Web application server.

Assuming a hardware failure, he will have to setup another machine, configuring and starting Apache and Tomcat. Besides the error-prone process, the lower bound of the mean time to repair is dependent on the time necessary for a human to react and reconfigure the failed system. With SELFWARE , the detection and recovery is automated.

The Mean-Time-To-Repair (MTTR) is dominated by the time to detect the failure, the time to redeploy the necessary software on the newly allocated node, and finally to restart legacy systems. To illustrate this, we manage a multi-tiered J2EE server running the RUBIS benchmark, using the Apache HTTP daemon and the Tomcat servlet engine. We provoked failures on either Apache or Tomcat, as depicted in Figure **??** and Figure **??**. SELFWARE detects and repairs the Apache daemon failure within 12 seconds and the Tomcat failure within less than 50 seconds. These numbers include the time for the failure detector to trigger and the time for downloading and installating the necessary software (Rubis, Apache daemon, and Tomcat). They include the installation of the Java wrappers and the apply of the overall management operations, including the writing of the configuration files from attributes. Ultimately, they also include the time it takes for Apache or Tomcat to start. While Apache is a fast starter, Tomcat is rather slow. While these numbers could be considered large, they are orders of magnitude better than any manual repair time, even by skilled operators.

We know that SELFWARE can be applied to a clustered J2EE where Apache can be used with the *modJK* plugin that can load balance requests on multiple remote Tomcats. This provides high-availability with respect to Tomcat failures. SELFWARE can repair failed Tomcat instances while maintaining the high availability of the Web server. This is true because wrappers may actually delay and re-order management operations within the commit of a repair session. For instance, the Apache daemon is kept running throughout the commit of the repair session, that is, while we repair the failed Tomcat. It is only stopped

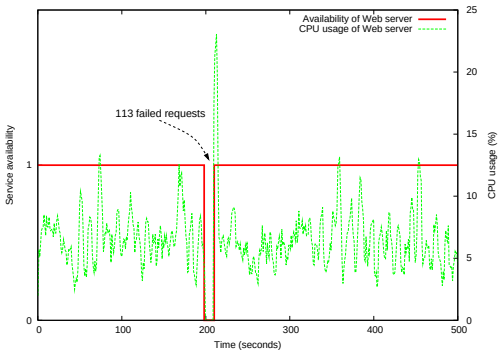

Fig. 7: Apache failure

figure

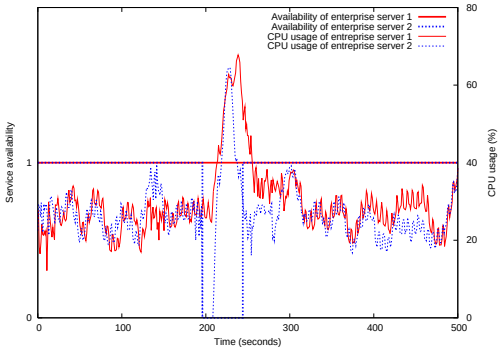

Fig. 8: Tomcat failure

figure

and restarted by its wrapper at the very last moment—upon receiving the end-commit. The interruption of service will be down to the time it takes to restart the Apache HTTP daemon. This restart is required by the *modJK* plugin that requires to be stopped to be reconfigured. For high-end availability, a modified plugin is something that can be done. The overall point is that SELFWARE provides safe and automated repair without hindering the legacy system performance, being compatible with clustered legacy systems tuned for high availability.

Finally, we experimented with the self-self-repair behavior of Selfware itself and its overhead on the ability of SELFWARE to repair managed legacy systems. We kept the above failure of a Tomcat but forced a simultaneous failure of one of the SELFWARE replicas (including both a replica of the repair manager and the managed architecture). These three failures are detected and handled in this experiment in one repair session. Hence, there is more work to do for repairing not only the lost Tomcat but also the lost replicas of the repair manager and the managed architecture. As above, the repair of Tomcat and of SELFWARE can be done without impacting the availability of the Web server (but for the short restart of the Apache daemon if we use an unmodified *modJK*). The most important point is that SELFWARE remains 100% available even facing partial failures of SELFWARE replicas. A repair of a failed replica of either the managed architecture or the repair manager is done entirely in the background and introduces no disruption in the ability of SELFWARE to repair any managed system.

VII. RELATED WORK

In section A, we consider autonomic management systems that deal with legacy applications. Section B and C focus on architecture-based management frameworks, that mostly rely on a formal or semi-formal description of the managed system structure typically expressed in terms of components and bindings using an Architecture Description Language (ADL). In section B, we present architecture-based management frameworks based on nun-reflective components models. Such frameworks have to ensure the causal consistency between the description of the system architecture and the effective structure of the system at runtime. Finally, section C describes architecture-based management frameworks based on reflective component model.

A. Legacy management framework

Many autonomic management frameworks for legacy systems are based on ad-hoc solutions that are tied to particular context. This reduces the reusability of the management services and policies, that need to be reimplemented each time a new legacy system is introduced in the system. This trend is well illustrated in the context of Internet Services where a lot of projects provides ad-hoc solutions for self-healing or self-optimization concerns.

For instance, [24], [2], and [16] have considered the management of a dynamically extensible set of resources in the context of Internet services. In [21], [22], the authors propose a self-optimized dynamic provisioning algorithm that specifically targets a cluster of databases. [18] describes a solution to provide adaptation to changing workloads specifically for Web servers. In the same way, the JAGR project [6] provides a solution for self-recoverability in the context of Enterprise Jave Beans running into the JBoss Application Server.

Apart from the previous frameworks, other systems like Rainbow [8] and KX [17] propose a solution that can be used to manage different legacy systems by retrofitting autonomic computing onto such systems without modifying the legacy code. KX runs as a decentralized set of loosely coupled components communicating via a publish/subscribe mechanism. These components correspond to sensors (watching the system), gauges (aggregating the sensor data), controllers (making decision) and effectors (reconfiguring the system). Whereas gauges and controllers are generic components that can be reused over a range of systems, sensors and effectors are, as in Selfware, wrappers components tightly coupled to the target system.

B. Framework based on nun-reflective component models

Rainbow [8], [11] and Darwin [12] are well representative of frameworks based on nun-reflective component models.

Rainbow [8] is an architecture-based management framework that supports self-adaptation of software systems. It uses an abstract architectural model to monitor an executing system's runtime properties, evaluates the model for constraints violations, and if a problem occurs, performs global adaptations of the running system. One main objectif of Rainbow is to favorize the reusability of their framework from one system to another, by dividing the framework into a generic system layer composed of probes and effectors, and a specific architecture layer defining the constraints, rules and strategies for adaptation. A translation service is used to manage the mapping between the system layer to the architecture layer and vice versa.

In Selfware, we only consider one layer of management, that is composed of reflective components representing the managed elements and providing, by reification, an architectural model of the runtime system. Selfware is moreover inherently distributed, while the Rainbow framework is based on a centralised design. Finally, Rainbow concentrates on the system adaptation in terms of autonomic policies and event processing - it does not address the deployment issues and the self-management of the management system (self-self management).

[11] proposes a framework for creating architecture-based autonomic systems. Event-based software architectures are targeted, because the managed software elements are loosely coupled (an element can be replaced without impacting the other elements). The architecture of a managed system is represented in xADL, an extensible, XML-based ADL. Changes to software architectures, such as an architectural repair, are represented as architectural differences, also expressed in a subset of the xADL langage. The framework is composed of a specific component, called an architecture evolution manager, that can instanciate and update a running system whenever its architectural description change. This component is thus responsible for managing the mapping between the running system and its architectural description. As in [8], this approach requires a mapping between the architectural description and the running system, which is automatic with Selfware. Futhermore, the aspects related to the reliability of the components are presented as a future work, and the self-self-management has not been taken into account.

Darwin [12] proposes a component model based on an explicit architectural specification expressed in the alloy langage [13]. Components are associated with constraints that define their behavior according to the architectural evolution of the global system. These constraints drive the autonomic behavior of components, providing them with self-organising properties and self-configuring bindings. At runtime, each component contains an implementation, a manager and a configuration view. This view can be seen as a checkpoint of the current architectural state of the *global* system. A component manager maintains the consistency of its configuration view through the use of a group protocol, which tolerates the failure of individual components. It also adjust the component's configuration in accordance with the configuration view.

The component model proposed by [12] targets more specifically self-organising systems that allow components to control their configuration in a decentralized manner. This motivates the use of a globally replicated architectural view, that could become an issue when the number of managed components becomes high. While the self-organising properties of this component model are interesting for getting autonomic capabilities, it could not be directly used as a base for a management framework as Selfware, because it would imply to provide a decentralized design of the autonomic managers, in order to embed a copy of them within each component. Supporting such a design without involving a strong coordination between the managers is a big challenge as our autonomic managers often take global decisions concerning more than individual components.

C. Reflexive component framework

With reflective component models, managed systems are implemented as a collection of interconnected components enhanced with a meta-level providing introspection and reconfiguration capabilities on the components structure. The meta-level directly provides a causally connected representation of the component structure, mainly by ensuring that any changes performed on the component structure at the meta-level are reported at the base-level.

[4] considers the use of reflective middleware to develop self-managing systems as a challenging research direction. Our work on Selfware falls in this category of systems, as other projects such as OpenORB [10], Plastik [3].

OpenORB [10] is a middleware platform built around a well-founded reflective lightweight component model, called OpenCOM. Like the Fractal reflective component model used in Selfware, the OpenCOM runtime provides support for a specializable and extensible meta-level providing introspection and reconfiguration operations on components. By managing the adaptability of a distributed architecture at the level of the component model, the OpenORB platform aims to provide a built-in support for building highly flexible distributed architectures ensuring reconfiguration integrity. Compared to our work, authors of OpenORB state that one needs a reflective component based middleware to build autonomic management system on top of it, but they do not further investigate the mechanisms and policies in such an autonomic system and do not adress the self-self-management of the plateform despite the capabilities provided by reflective components.

The reflective OpenORB platform is used in the Plastik infrastructure [3], that follows an architecture-based management approach relying on the reified architecture provided by OpenORB. Plastik focus on constraints and general invariants that can be associated to the specification of a component-based system, through the notion of *architectural styles*. Any component reconfiguration is accepted as long as the invariants defined in its associated architectural style are not violated. This approach, as well as those used in [9], conforms to a design pattern proposed by [20], which exposes architectural style requirements for building self-managed systems.

In Selfware, we did not consider the notion of architectural styles because a main motivation of our work was to provide a way to build generic autonomic managers, able to manage any kind of managed element independantly of its architectural style.

It is important to point out that this genericity aspect is a strong base for the recursive design of Selfware. Indeed, it provides Selfware with self-self-management capabilities by allowing the Selfware components to be managed as any other managed components. Neither Plastik nor FORMAware consider the issue of the infrastructure self-management.

However, our experience gained through the experimentations described in this paper has shown that the management task may be simplified by allowing to define semantically higher level reconfigution operations. In other words, architectural styles could be used complementary to the generic low-level management functions of Selfware, as a way of (1) constraining the dynamic evolution of an architecture; and (2) providing higher level management functions. We believe that a powerful approach would indeed consist in combining generic autonomic managers dealing with basic management functions (e.g., deployment, repair) with more specific ones that take into account both the functionnal and nun-functionnal specificities of the managed elements.

VIII. Conclusion

This paper describes SELFWARE , a reflexive component-oriented framework for building autonomic managers in the architecture-based paradigm. We report here our experience in building autonomic managers for self-protection, self-optimization, and self-repair.

At the heart of the SELFWARE framework is the reflexive architecture that is captured by meta-level controllers. Only five such controllers are defined by SELFWARE thereby defining a small and uniform management interface for developing autonomic managers. The advocated programming model for autonomic managers is centered on observing the architecture and atomically reconfiguring it. The reflexive architecture captures relevant aspects of the managed distributed system, not only the managed elements of applications but also the hardware and software elements of the underlying distributed systems hosting these applications.

We adopted a recursive design for SELFWARE , meaning that autonomic managers and all SELFWARE middleware components are designed as SELFWARE components. This is a very important design decision since it allows SELFWARE to self manage itself. The importance of this design decision shows through both self-protection and self-repair that are both recursive concerns. Indeed, a self-protection manager may be itself the target of attacks and therefore needs to self protect. The same is true for a self-repair manager that may experience failures impacting its own

functioning and therefore needs to self repair. Both designs were greatly simplified by SELFWARE recursive design. We therefore argue that a reflexive component-oriented framework, combined with both atomic reconfiguration and transparent component replication, is a promising direction for building future autonomic systems.

SELFWARE allows to build such autonomic systems either from scratch with new component technologies or leveraging wrapped legacy systems. Regarding the wrapping of legacy systems, our experience is also a positive one. Our approach allows most of the complex management tasks to be achieved in very generic ways, mostly independent of any specifics of legacy systems. More work is needed to assess the appropriateness of new component technologies to autonomic architecture-based management. Capturing the entire architecture sometimes proves difficult and dynamic reconfigurations are often a challenge for most component-oriented runtime.

REFERENCES

[1] C. Amza, E. Cecchet, A. Chanda, A. Cox, S. Elnikety, R. Gil, J. Marguerite, K. Rajamani, and W. Zwaenepoel. Specification and Implementation of Dynamic Web Site Benchmarks. In *IEEE 5th Annual Workshop on Workload Characterization (WWC-5)*, Austin, TX, November 2002.

[2] Karen Appleby, Sameh A. Fakhouri, Liana L. Fong, German S. Goldszmidt, Michael H. Kalantar, S. Krishnakumar, Donald P. Pazel, John A. Pershing, and B. Rochwerger. Oceano-SLA based management of a computing utility. In *Proceedings of Integrated Network Management*, pages 855–868, 2001.

[3] T. Batista, A. Joolia, and G. Coulson. Managing Dynamic Reconfiguration in Component-based Systems. In *European Workshop on Software Architectures*, Pisa, Italy, June 2005.

[4] G.S. Blair, G. Coulson, and P. Grace. Research directions in reflective middleware: the Lancaster experience. In *Proceedings of the 3rd Workshop on Adaptive and Reflective Middleware (ARM'04)*, New York, NY, USA, 2004. ACM Press.

[5] É. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.B. Stefani. The Fractal Component Model and its Support in Java. *Software – Practice and Experience (SP&E)*, 36(11-12):1257–1284, September 2006. special issue on "Experiences with Auto-adaptive and Reconfigurable Systems".

[6] George Candea, Emre Kiciman, Steve Zhang, Pedram Keyani, and Armando Fox. Jagr: An autonomous self-recovering application server. In *Active Middleware Services*, pages 168–178, 2003.

[7] E. Cecchet, J. Marguerite, and W. Zwaenepoel. C-JDBC: Flexible Database Clustering Middleware. In *USENIX Annual Technical Conference, Freenix track*, Boston, MA, June 2004. http://c-jdbc.objectweb.org/.

[8] Shang-Wen Cheng, An-Cheng Huang, David Garlan, Bradley Schmerl, and Peter Steenkiste. Rainbow: Architecture-Bases Self Adaptation with Reusable Infrastructure. *IEEE Computer*, 37(10):46–54, October 2004.

[9] S.W. Cheng, D. Garlan, B. Schmerl, J.P. Sousa, B. Spitznagel, and P. Steenkiste. Using Architectural Style as a Basis for Self-repair. In *Proceedings of the 3rd Working IEEE/IFIP Conference on Software Architecture (WICSA 2002)*, Montreal, 2002.

[10] Geoff Coulson, Gordon S. Blair, Michael Clarke, and Nikos Parlavantzas. The design of a configurable and reconfigurable middleware platform. *Distributed Computing*, 15(2):109–126, 2002.

[11] E.M. Dashofy, A. van der Hoek, and R.N. Taylor. Towards Architecture-based Self-Healing Systems. In *Proceedings of the First ACM SIGSOFT Workshop on Self-healing Systems*, Charleston, 2002.

[12] I. Georgiadis, J. Magee, and J. Kramer. Self-organising software architectures for distributed systems. In *1st Workshop on Self-Healing Systems (WOSS'02)*, New York, NY, 2002.

[13] D. Jackson. Alloy: A lightweight object modelling notation. In *MIT Lab for Computer Science*, July 1999.

[14] J.O. Kephart and D.M. Chess. The Vision of Autonomic Computing. *IEEE Computer Magazine*, 36(1), 2003.

[15] MySQL Web Site. http://www.mysql.com/.

[16] James Norris, Keith Coleman, Armando Fox, and George Candea. OnCall: Defeating spikes with a free-market application cluster. In *1st International Conference on Autonomic Computing (ICAC'04)*, pages 198–205, New York, NY, USA, May 2004.

[17] Janak J. Parekh, Gail E. Kaiser, Philip Gross, and Giuseppe Valetto. Retrofitting autonomic capabilities onto legacy systems. *Cluster Computing*, 9(2):141–159, 2006.

[18] Prashant Pradhan, Renu Tewari, Sambit Sahu, Abhishek Chandra, and Prashant Shenoy. An observation-based approach towards self-managing web servers. In *IWQoS'02: Tenth IEEE International Workshop on Quality of Service*, pages 13–22, 2002.

[19] Apache HTTP Server Project. Apache. http://httpd.apache.org/.

[20] M.M. Rakic, N. Mehta, and N. Medvidovic. Architectural style requirements for self-healing systems. In *Proceedings of the First Workshop on Self-healing Systems (WOSS'02)*, New York, NY, USA, 2002. ACM Press.

[21] Gokul Soundararajan and Cristiana Amza. Autonomic provisioning of backend databases in dynamic content web servers. Technical report, Department of Electrical and Computer Engineering, University of Toronto, 2005.

[22] Gokul Soundararajan, Cristiana Amza, and Ashvin Goel. Database replication policies for dynamic content applications. In *First EuroSys Conference (EuroSys 2006)*, Leuven, Belgium, April 2006.

[23] The Apache Software Foundation. Apache Tomcat. http://tomcat.apache.org/.

[24] Bhuvan Urgaonkar and Prashant J. Shenoy. Cataclysm: policing extreme overloads in internet applications. In *Proceedings of the 14th international conference on World Wide Web, (WWW'05)*, pages 740–749, Chiba, Japan, May 2005.