From Autonomic to Self-Self Behaviors: The JADE Experience

SARA BOUCHENAK, FABIENNE BOYER, BENOIT CLAUDEL, NOEL DE PALMA, OLIVIER GRUBER, and SYLVAIN SICARD, University of Grenoble

Autonomic computing enables computing infrastructures to perform administration tasks with minimal human intervention. This wrap-up paper describes the experience we gained with the design and use of JADE—an architecture-based autonomic system. The contributions of this article are, (1) to explain how JADE provides autonomic management of a distributed system through an architecture-based approach, (2) to explain how we extended autonomic management from traditional self behaviors such as repairing or protecting a managed system to self-self behaviors where JADE also fully manages itself as it manages any other distributed system, (3) to report on our experience reaching self-self behaviors for two crucial autonomic properties, repair and protection.

Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed applications*

General Terms: Reliability, Security

Additional Key Words and Phrases: Autonomic computing, architecture-based management, JEE

ACM Reference Format:
Bouchenak, S., Boyer, F., Claudel, B., De Palma, N., Gruber, O., and Sicard, S. 2011. From autonomic to self-self behaviors: The JADE experience. ACM Trans. Auton. Adapt. Syst. 6, 4, Article 28 (October 2011), 22 pages.
DOI = 10.1145/2019591.2019597 http://doi.acm.org/10.1145/2019591.2019597

1. INTRODUCTION

The goal of autonomic computing [Kephart and Chess 2003] 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. First, the administration costs are an increasing part of the total information system costs. Second, the difficulty of the administration tasks tends to reach the limits of what human administrators can handle. In this context, the self-management capabilities enabled by autonomic computing provide powerful answers in matters such as self-configuration, self-optimization through continuous performance monitoring, self-repair through detecting and repairing failures, and self-protection through detecting and defending against malicious attacks.

In JADE, we provide autonomic management for loosely coupled distributed systems through an architecture-based approach. First, different autonomic managers observe and monitor the managed system through its architecture. Second, based on these observations, each autonomic manager may take appropriate steps to maintain the managed system within preset goals. In this approach, different autonomic managers

Authors' address: S. Bouchenak, F. Boyer, B. Claudel, N. De Palma, O. Gruber, and S. Sicard; email: {sara.bouchenak, fabienne.boyer, benoit.claudel, noel.depalma, olivier.gruber, sylvain.sicard}@inrialpes.fr.

DOI 10.1145/2019591.2019597 http://doi.acm.org/10.1145/2019591.2019597

are responsible for different autonomic properties, such as self-repair, self-protection, or self-optimization. It is our experience that a reflective component-oriented design is very effective for building such self-management capabilities.

JADE uses components to capture the traditional concept of managed elements as well as the physical machines hosting them. Managed elements are wrapped as fractal components [Bruneton et al. 2006] that provide simple but powerful control operations. These control operations are the basis of the JADE uniform management interface, enabling distributed heterogeneous legacy systems to be managed remotely. With this wrapping in place, the overall managed system appears as a set of distributed and interconnected components.

JADE captures this component-oriented architecture and thereby provides autonomic managers with the ability to observe and manipulate the managed system. Through introspection, autonomic managers can observe not only the architecture of the managed distributed system but also its runtime behavior, including for instance, security-related communication patterns for detecting intrusion. Through reconfiguration, autonomic managers can manipulate the architecture of the managed distributed system, including for instance, the ability to provide higher availability through replicating components across nodes.

A fundamental challenge that is addressed by JADE is self-self management, meaning that JADE is able to entirely manage itself as it manages any other distributed systems. Reaching self-self management is essential for ensuring the overall reliability of an autonomic management system. Without it, the management system guarantees the reliability of the managed system, but nothing guarantees the reliability of the management system itself. In particular, the autonomic management system can fail or be the target of attacks. Based on a unique recursive design, JADE self-protects and self-repairs.

This article is organized as follows. In Section 2, we present the main principles of the JADE design. In Section 3, we explain how we reach self-self behaviors through enhancing the JADE design with recursion and replication principles. In Sections 4 and 5, we discuss self-self behaviors on the essential and challenging properties of autonomic repair and protection. In Section 6, we give results about the autonomic management of advanced Web servers (JEE). In Section 7, we discuss related work. In Section 8, we discuss the lessons learned from the JADE project and we conclude.

2. JADE DESIGN

The design of JADE is best described in these steps. First, we introduce the concept of *wrappers*: components that wrap legacy systems in order to provide a uniform set of management operations across heterogeneous legacy systems. Second, we discuss the distributed nature of the managed system: JADE manages loosely-coupled legacy systems, distributed across networked machines. Third, we introduce the concept of the *managed architecture*, which captures the distributed architecture of the managed system in a single component-oriented data structure.

2.1. Wrapping Legacy Systems

Wrapping legacy systems is the first step towards autonomic management of legacy systems. In JADE, any managed legacy system is wrapped as one FRACTAL component. In FRACTAL, each component offers a small but powerful set of control interfaces that provides the core management operations to JADE. Each wrapper is therefore in charge of implementing these control interfaces, in legacy-specific ways.

In Sicard et al. [2008], we discussed the full FRACTAL model and the corresponding reflective requirements that a component model must have in order to support

autonomic management. The three most important control interfaces implemented by wrappers are the lifecycle controller, the attribute controller, and the binding controller.

— The *lifecycle controller* is about starting and stopping the wrapped legacy system. The implementation of this controller is usually straightforward, leveraging available start and stop scripts.
— The *attribute controller* captures, as key-value pairs, the configuration data of the wrapped legacy system. Hence, legacy systems can be configured in a uniform way through setting attribute values. An example of one such attribute is the port used by an Apache daemon to listen to incoming HTTP requests. Most often, wrappers implement the attribute controller by direct manipulation of the configuration files of the legacy systems they wrap.
— The *binding controller* captures the presence of communication channels between legacy systems. For instance, wrapping the Apache HTTP daemon, a binding captures the TCP/IP connection between the HTTP daemon and a servlet engine such as Tomcat. It is important to point out that bindings are in between FRACTAL components (the wrappers) and only capture the existence of communication channels between the wrapped legacy systems. In other words, bindings are not involved in the actual communication; wrapped legacy systems communicate directly.

A constraint that should be considered when programming wrappers is that they are expected to be fail-stop, a necessary property for ensuring the reliability of the JADE management as explained later in this article. This requires not only wrappers to be fail-stop themselves but also that failing wrappers actually stop their wrapped legacy system before they fail. It is our experience that such fail-stop assumptions are realistic and an important requirement to build self-* properties.

Despite fail-stop assumptions, most wrappers are extremely simple. All wrappers that we wrote so far are direct programmatic transcriptions of what human administrators regularly do with scripts and a console.

2.2. Distributed Managed System

JADE targets loosely coupled legacy systems built as an assembly of legacy subsystems. Such systems are representative of today's distributed systems, such as multitiered Web application servers, Web services, and message-oriented middleware. More traditional operating systems also rely on loosely coupled subsystems, such as NFS, DNS, printer spoolers, or email systems.

Providing autonomic capabilities for such distributed systems suggests wrapping legacy subsystems with components that offer a uniform set of management operations. Wrappers are fractal components that are distributed across machines in the same way the managed legacy subsystems are. Wrappers are colocated with their legacy systems because most of the management operations they provide make calls to legacy scripts. Through wrapper components, JADE builds the distributed managed system (DMS), as depicted in the intermediate level of Figure 1. The DMS provides the JADE autonomic managers with a uniform and complete model of the distributed system they manage.

JADE also captures the administered physical machines as fractal components. There is one node component per physical machine known to JADE in the managed system. In some sense, we can say that a node component wraps a physical machine. In particular, a node component provides the ability to deploy and undeploy FRACTAL components. Therefore, a node component has the knowledge of the wrappers that are deployed on the physical machine it wraps.

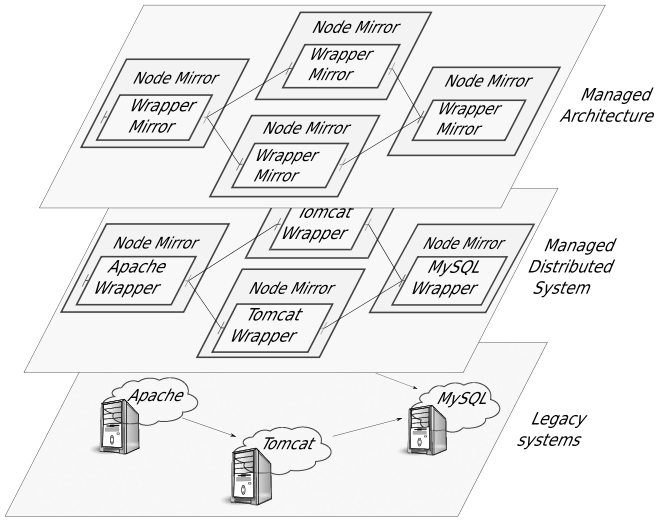

Fig. 1. Jade overall architecture.

A node component also supports runtime probes, such as intrusion detection or failure probes. Some probes are legacy subsystems that must be wrapped with FRACTAL components while other probes may be directly implemented in Java as components. Probes provide autonomic managers with crucial runtime information about legacy systems, such as detecting failures for the self-repair manager or detecting intrusions for the self-protection manager.

2.3. Managed Architecture

The managed architecture, depicted in Figure 1, provides autonomic managers with a mirror view of the distributed managed system (DMS). For each component in the DMS, the managed architecture has a mirror component that provides the very same controllers as the DMS component. In other words, a mirror has the same lifecycle state (started or stopped) and the same attributes as the component it mirrors. Moreover, mirrors have bindings between them that are isomorphic to the bindings between the components they mirror.

Through mirrors, autonomic managers can both introspect and reconfigure the architecture of the managed system, shielded from its distributed nature and its failures. 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 started or stopped as well as the bindings that link them. Managers can also know which wrapper is deployed where, through the mirrors of node wrappers that capture the knowledge of locally installed wrappers.

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 some of their attributes. They can also remove or create bindings between wrappers. Autonomic managers therefore reconfigure the managed system through a sequence of management operations invoked on mirrors, such as *start()*/*stop()*, *bind()*/*unbind()*, and *setAttribute()*.

Reconfigurations happen on managed architecture through atomic sessions. At commit time of a session, JADE will apply the reconfiguration done in the managed architecture onto the distributed managed system. In some sense, JADE replays at commit

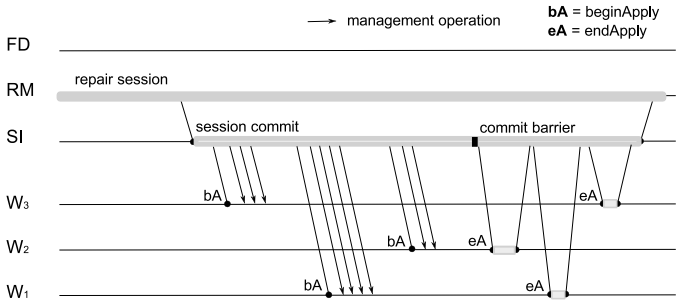

Fig. 2. Session commit.

time the management operations of the atomic session on the corresponding wrappers. In turn, wrappers will apply the corresponding management operations onto the legacy systems they wrap. This is globally a two-phase commit process, as depicted in Figure 2. During the first phase, the commit reconfigures the wrappers of the impacted mirrors, invoking the necessary management operations onto these wrappers. During the second phase, the commit enters a barrier, waiting for all wrappers to effect these management operations on the subsystem they wrap.

For each wrapper, all invoked management operations are bracketed by a *beginApply* and an *endApply*. In other words, JADE always invokes the *beginApply* method on a wrapper before it invokes any management operations. Once all management operations have been invoked, JADE invokes the *endApply* method. Across wrappers, JADE globally orders the *endApply* invocations following the startup ordering constraints in the managed architecture. This means that when a startup ordering constraint implies that a legacy system managed by a wrapper, W_i, be started after a legacy system managed by a wrapper, W_j, any repair action of a wrapper, W_j must happen before the repair of wrapper W_i if both W_j and W_i are detected as failed. By the time all *endApply* invocations on wrappers have returned, the commit is finished and the current reconfiguration session completed.

It is important to realize that at commit time, the reconfiguration has already been successful on the managed architecture, meaning that mirrors have been completely and correctly reconfigured. Failures that shall be considered are therefore about failures during the commit itself, when applying the reconfiguration onto wrappers and ultimately onto legacy systems. Such failures are detected, but do not prevent, the commit from completing successfully. For instance, if a reconfiguration operation on a wrapper is unsuccessful, the wrapper fail-stops itself. Such failure will be detected by JADE and will be automatically repaired in a followup but separate repair session. In case the failure of a wrapper W impacts other wrappers, these wrappers will also fail-stop and be later repaired.

3. SELF-SELF DESIGN

JADE not only models and manages a distributed system, it is itself a distributed system. This suggests that JADE can manage itself in order to guarantee the reliability of its autonomic management. We tackled this challenge following a recursive design.

3.1. Recursive Design

The basis of our recursive design is that JADE is entirely designed and implemented using the very same component model that was used for programming wrappers. This means that the autonomic managers and the managed architecture are designed as

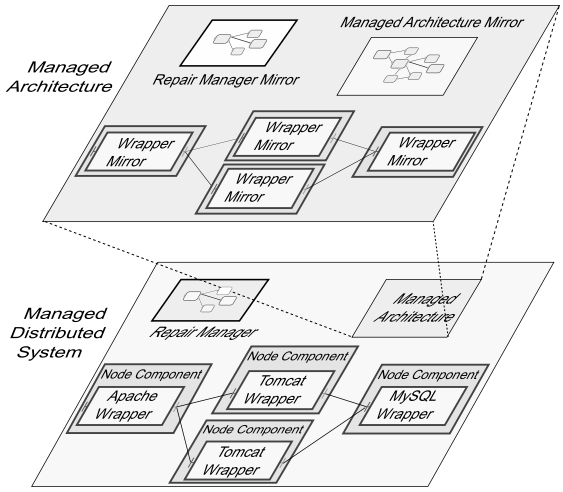

Fig. 3. Recursive architecture.

FRACTAL components, implemented in Java. Thus, the JADE components appear in the distributed managed system, alongside other wrappers and node components, and they are are mirrored in the managed architecture, as any component in the DMS is. This is depicted in Figure 3. The lower plane shows the distributed managed system, which includes the managed elements now including JADE components such as the repair manager and the managed architecture. The top plane represents the mirrors in the managed architecture, which include mirrors for the JADE components, including the mirrors of the JADE components used to implement the managed architecture. The recursion stops there as we do not mirror mirrors.

The recursive nature of our design is now apparent—the managed architecture mirrors itself. With this recursive design, autonomic managers can observe and reconfigure either the managed legacy systems or the internals of the JADE system itself.

3.2. Replicated Design

A recursive design is only a first step toward the self-self behaviors; it creates the possibility for JADE to manage itself using the very same techniques it applies on any managed systems. However, when a JADE component becomes unavailable because of a failure or an attack, the ability of JADE to manage itself may be impaired. This is especially true if the repair manager fails, as it obviously cannot repair itself anymore. The same is true for the protection manager.

To ensure the fault-tolerance and high-availability of JADE, we replicate JADE on a cluster called the JADE cluster, as described in Figure 4. In particular, we replicate the managed architecture and the core autonomic managers requiring self-self properties, such as the repair and protection managers. We choose to replicate JADE using an active replication mechanism [Guerraoui and Schiper 1996], since JADE components are deterministic. Consequently, each JADE component becomes a replicated Java object and Java remote unicast references are no longer sufficient. Indeed, for transparency reasons, a reference between two replicated components shall become a reference between two groups of Java objects.

We therefore extended the Java Remote Method Invocation (RMI) with groupcast references, which support remote method invocations between replicated components. A groupcast reference combines the semantics of a multicast and a gathercast, as

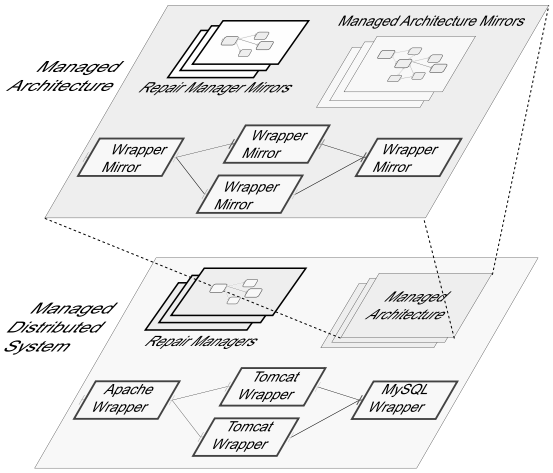

Fig. 4. Replicating JADE.

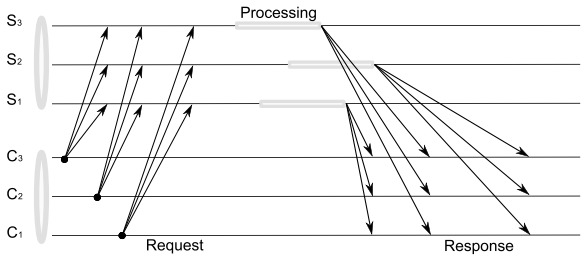

Fig. 5. Multicast-gathercast binding.

illustrated in Figure 5. This groupcast semantic has been implemented on Jgroups,[1] a toolkit that provides group membership services and a reliable totally-ordered message delivery to dynamic groups. We allocate one group per replicated component, where each group is identified by a globally unique identifier (GUID) that is stored in the managed architecture.

Using groupcast references, we can introduce replication transparently. Although each component may be replicated at its own cardinality and across its own set of hardware nodes, JADE managers perceive nonreplicated components being connected through unicast bindings. Between replicated components however, unicast bindings are in reality, groupcast bindings using groupcast references. This translates into the combination of the following two protocols.

Each incoming binding to a replicated component translates to a multicast semantics onto the different replicas. All correct replicas independently and concurrently execute the methods invoked on the component. Hence, if a fault prevents one replica from operating correctly, the other replicas will produce the required results without the delay required for recovery.

Each outgoing binding from a replicated component translates to a gathercast semantics, ensuring a once-and-only-once semantics. Since replicated components may invoke methods on nonreplicated components, the gathercast detects and suppresses duplicate requests that are generated by the different replicas based on message

[1]http://www.jgroups.org/javagroupsnew/docs/index.html

identifiers. This is especially important for absorbing the redundant management operations that an active replication scheme introduces.

4. AUTONOMIC REPAIR

This section discusses autonomic repair. In most management systems, an autonomic repair means the ability to repair a managed system. This is often called self-repair even though the failures of the management system are not detected and repaired, requiring a manual intervention from an operator. For JADE, an autonomic repair not only includes the ability to repair a managed system but also the ability to repair itself, which we call self-self repair to contrast with the traditional self repair denomination.

4.1. Self Repair

Our autonomic repair is architecture-based and as incremental as possible, avoiding any shutdown of the overall managed system when repairing the failures of individual subsystems. After detecting a failure, our self-repair manager analyzes the failures by introspecting the managed architecture and repairs these failures by reconfiguring this architecture. Our self-repair manager handles fail-stop failures of either nodes or subsystems.

During the analysis step, the repair manager identifies the impacts of the detected failure, determining the set of failed components that were lost due to the node failure. By introspecting the node mirror representing the failed node in the Managed Architecture, the self-repair manager is able to discover all the components that were deployed and running on this failed node. Indeed, the managed architecture protects the architectural knowledge of the managed system that would otherwise be lost to failures. Thus, introspecting the managed architecture, our repair manager can know which wrappers have been lost to a failed node as well as the complete architectural state of lost wrappers, including their attribute values and their bindings.

From there, the repair process is essentially a three-step process. One, it substitutes a failed node with a new one from a pool of available hardware nodes. Two, it redeploys on that new node, the lost wrappers and their wrapped legacy systems. Three, it fully reconfigures the redeployed wrappers, which in turn fully reconfigure the wrapped legacy systems. This last step includes cleaning up stale bindings before creating correct ones, which requires computing the set of impacted components.

Impacted components are all the components currently bound to a failed component. The cleaning up of stale bindings is simply done by unbinding them from impacted components in the managed architecture. At commit time, JADE forwards the *unbind()* operations to the corresponding wrappers that will request their wrapped legacy to close stale communication channels. The creation of new bindings is also done on the managed architecture. Again, at commit time, JADE forwards these *bind()* operations to wrappers, allowing them to inform their legacy of the new communication channels to use.

A typical example of this situation can be sketched between an Apache HTTP daemon and its Tomcat servlet engines. When the hardware node where a Tomcat servlet engine runs, fails, a new instance of a Tomcat servlet engine must be recreated on a new hardware node. Therefore, the Apache HTTP daemon must first close its socket to the failed Tomcat and reopen one to the newly created Tomcat. The *unbind()* operation resets the IP address and port in the Apache configuration file while the *bind()* operation sets the new correct values. It is interesting to point out that the Apache daemon has to be restarted to reread its configuration file. Hence, to apply the unbind-rebind operations, the wrapper has to actually shut down and restart the Apache HTTP daemon. Other legacy systems have a more dynamic approach and can be reconfigured

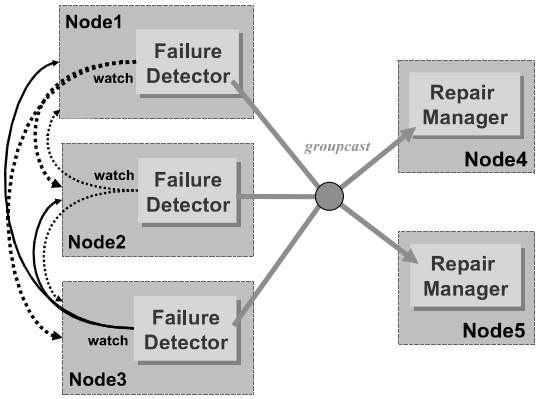

Fig. 6. Replicated failure detector.

without having to stop and restart. In any case, it is the responsibility of the wrappers to respect such reconfiguration constraints imposed by the wrapped legacy system.

4.2. Self-Self Repair

Our self-self repair is obtained as follows. First, we replicate all the components involved in the autonomic repair. This means replicating the repair manager, the managed architecture, and the fault detector. Second, each replica of the repair manager watches over all the components mirrored in the managed architecture. Third, given our recursive design, where JADE components are mirrored in the architecture, this means that each replica of the repair manager watches over all the replicas of JADE components. Hence, any failure of a JADE component, including those of the repair manager replicas, can always be repaired by the unaffected replicas of the repair manager.

This is a quite traditional use of active replication to reach high-availability and fault-tolerance. There is only challenge—we replicate our failure detector and failure detectors are known to be nondeterministic. Figure 6 illustrates our replication scheme for our failure detector; it is connected to the repair manager, itself a replicated component, through a groupcast reference. This dual replication ensures that the failure detector is no single point of failure and that all replicas of the repair manager are notified that a failure occurred. Each failure detector replica, called detector in this discussion, watches over all the hardware nodes managed by JADE, themselves watching over the fractal components they host. Each detector continuously reports a detected failure to the replicated repair manager until that failure is actually repaired or the repair manager decides, after several repair attempts, that the component has permanently failed.

Nevertheless, the nondeterministic nature of failure detectors is still a problem. The solution comes from our groupcast semantics, whose design forces a deterministic outcome. Indeed, our gathercast semantics absorb redundant messages, based on their sequence number and not their actual contents. Therefore, different failure reports multicasted with the same sequence number from different detectors will be absorbed. To the repair manager, it appears as if all detectors had multicasted the same failure report, hence the deterministic outcome. This design may enable a false positive to be absorbed but most importantly it introduces no false negative—no detected failure can remain unreported. Indeed, as long as a detector knows about a failure that has not been repaired, it will keep multicasting its failure report. At some point, either its

report will go through as it acquires the highest sequence number or another detector will report it. In either case, the failure will eventually be reported to our repair manager.

5. AUTONOMIC PROTECTION

This section discusses autonomic protection, another autonomic behavior, which applies to both the managed system and the management system. Self-protection 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, countermeasures can be put in place to contain its progression and the damages it creates. We focus on generic mechanisms that not only can recognize known and unknown attacks but are also independent from any specifics of wrapped legacy systems.

Our autonomic protection detects illegal communication channels using the knowledge of the managed architecture as the sense of self, which illustrates the importance of an architecture-based approach for autonomic protection.

In particular, this approach handles well, both self and self-self protection, where the protection manager itself is the target of attacks. Our sensors detect all the communication not explicitly authorized in the managed architecture. Hence, it is possible to react to all kind of attacks (known and unknown) using an illegal communication channel. For instance, it is possible to detect a port scanner and block the attack before the real intrusion. However, our approach does not prevent attacks that use legal communication channels.

5.1. Self-Protection

The assumption is that the managed architecture captures the knowledge of legal components and legal communication channels between these components. That is, components mirrored in the managed architecture represent legally installed software on hardware nodes. The bindings between mirrors represent legal communication channels. Since bindings capture the TCP/IP parameters used by underlying communication channels; they can be used to detect illegal communication channels. Indeed, any communication through a network connection that does not correspond 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 and legal architecture. To ensure this, all reconfigurations of the managed architecture are authenticated through asymmetric cryptography, making sure only official autonomic managers are allowed to reconfigure the managed architecture. This prohibits compromised components from manipulating the architecture and introducing illegal bindings between components, something that could allow them to authorize illegal communication channels.

Our protection mechanism relies on managed firewalls, one such firewall running on each node. In our prototype, we wrapped the netfilter firewall [Netfilter]. Firewalls are configured automatically from the self-knowledge available in managed architecture. Every time this architecture evolves, firewall configurations are updated accordingly. This is done by the self-protection manager, which observes the managed architecture and maintains the firewall configurations in sync using the knowledge of the communication ports and IP addresses of established communication channels between managed components.

When detected, illegal communications between nodes are prevented by firewalls that notify the self-protection manager. This detection is depicted in Figure 7 in the context of a multitiered JEE server, the firewall of node 4 detects an illegal

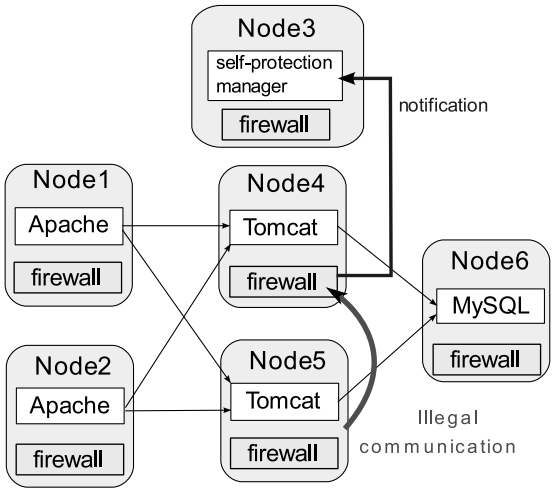

Fig. 7. Detection of an attack from node 5.

communication originating from node 5. Once an illegal communication is detected, different protection policies may be implemented. In our prototype, we chose to consider as compromised, the node from which the illegal communication originated. Our rationale is that only a compromised node in a JADE-managed system can attempt an illegal communication. The self-protection manager will therefore reconfigure the managed architecture through *unbind()* operations in order to isolate the compromised node.

5.2. Self-Self Protection

Given our recursive design in JADE, self-self protection can be provided by replicating the protection manager and the managed architecture. Since they are deterministic fractal components, our active replication scheme applies.

Intrusion detectors represented by wrapped firewalls need not be replicated since the availability of a firewall only needs to match the availability of the hardware node it runs on. Therefore, since our intrusion detectors are deterministic and not replicated, the reliable totally-ordered multicast is enough to ensure that all replicas of the protection managers receive intrusion events in the same order and all of them receive a given intrusion event or none of them do. The gathercast semantic is still required for a once-and-only-once semantics, avoiding wrapped firewalls to execute redundant configuration orders sent by the different replicas of the protection manager.

If a replica of the protection manager becomes infected and attempts to use an illegal communication channel, the attack will be detected by the other replicas of the protection managers. This will induce the isolation of the hardware node running the infected replica. However, we face the challenge of a malicious replica of the protection manager reconfiguring firewalls. Since each replica of the protection manager has the right to contact any wrapped firewall, this cannot be detected as an illegal use of a communication channel. To protect from this behavior, we used a specialized implementation of our groupcast protocol, introducing quorum voting in the gathercast semantics. When quorum voting is turned on, a method invocation is delivered to a wrapper if and only if a quorum of requests has been received. In our case, we set the quorum to the majority. Therefore, unless a majority of replicas of the protection manager requests a firewall reconfiguration, it will not be considered.

6. EVALUATION

For our experiments, we used the Java 2 Platform Enterprise Edition (JEE), which defines a model for developing Web applications. The architecture is classically divided in three tiers: the HTTP daemon (Apache), the application server (Tomcat), and the database tier (such as MySQL). Such applications receive requests from Web clients, route these requests through a Web server (provider of static contents), then to an application server to execute the business logic of the application, and finally to a database system that persistently stores data. Furthermore, to support high loads and provide higher availability of Internet services, a commonly used approach is the replication of each tier over a cluster.

Our testbed application is the RUBiS [Amza et al. 2002] application, a well-known JEE application benchmark based on servlets, which implements an auction site similar to eBay. The load injector of RUBiS emulates a variable number of clients sending a series of requests. It defines 26 Web interactions, such as registering new users, browsing, buying or selling items. For our experiments, we used two transition matrices, (1) the browse_only_transitions matrix (which contains only read-only operations) and (2) the default_transitions matrix (which contains 80% of read-only operations and 20% of write operations). This benchmarking tool gathers statistics about the generated workload and the Web application behavior.

We used the Rubis 1.4.2 version of the multitier JEE application running on several middleware platforms: Apache 1.3.29 as a Web server [Apache], Jakarta Tomcat 3.3.2 as an enterprise server,[2] MySQL 4.0.17 as a database server,[3] Tomcat clustering as the enterprise server clustering solution,[4] and c-jdbc 2.0.2 as the database server clustering system [Cecchet et al. 2004]. Experiments were performed using Linux, on IA-32 processor at 1.8GHz with 1GB of RAM, and connected via a 100Mb/s Ethernet LAN.

6.1. Autonomic Repair

Experiences with autonomic repair compare the case where Rubis is run and managed by JADE and when it is run and managed by hand. In these experiments, Rubis follows the default_transitions matrix. Without JADE, failures require the intervention of a human, who has to detect and understand them. Assuming a hardware failure, he has to set up another machine, configure, and start both Apache and Tomcat. Furthermore, human errors are considered as the root cause of roughly 20% to 50% of system outages [Gray 1986, 1990]. The lower bound of the mean time to repair is long, being dependent on the time necessary for a human to react and reconfigure the failed system. With JADE, 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 the time to restart the legacy system.

To evaluate this, we forced failures on either the Apache HTTP daemon or the Tomcat servlet engine, as depicted in Figure 8 and Figure 9. JADE 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 installing the necessary software (Rubis, Apache daemon, and Tomcat). They include the installation of the Java wrappers and applying the overall reconfiguration operations, including the writing of the configuration files from attributes. Ultimately, they also include the time it takes for Apache or Tomcat

[2]http://www.tomcat.apache.org/
[3]http://www.mysql.com/
[4]http://www.tomcat.apache.org/

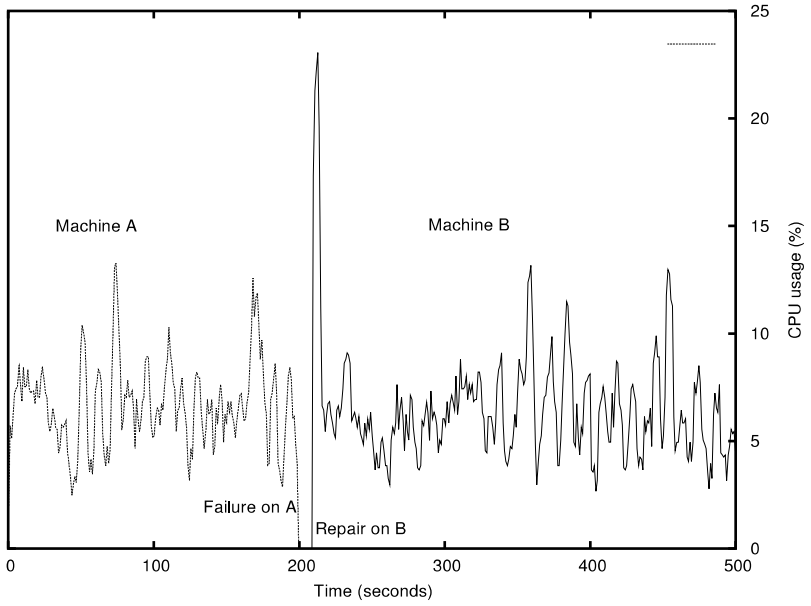

Fig. 8. Apache Web server failure and recovery.

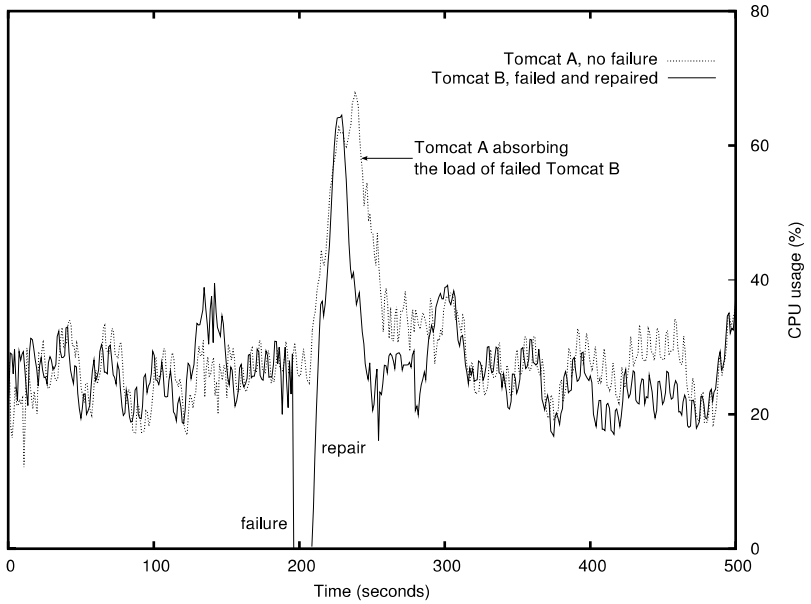

Fig. 9. Tomcat failure and recovery.

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. Indeed, manual repairs are the largest contributor to MTTRs that are several hours long on average [Kalyanakrishnam et al. 1999; Oppenheimer et al. 2003].

JADE has also been applied to a clustered JEE, where Apache is used as a load-balancer to balance requests on multiple remote Tomcats. While such clustered JEE provides high availability with respect to Tomcat failures, maintaining the replicas' cardinality despite failures, still require the intervention of human administrators. Using JADE, failed Tomcat instances can be automatically repaired while maintaining the high availability of the Web server. The challenge is that the load-balancer does not support hot replacement of Tomcat replicas. In other words, to change the configuration of the load-balancer—that is, remove the failed replica of a Tomcat and add the newly created replica—our Apache wrapper must start and stop the Apache HTTP daemon it wraps.

Fortunately, our approach allows the wrapper of the Apache HTTP daemon to reorder management operations, updating the load-balancer configuration while keeping the HTTP daemon up and running, serving HTTP requests using the Tomcat replicas that are still available. Our wrapper will do only a quick stop-start sequence on the HTTP daemon at the end of the commit of the reconfiguration. Since the Apache HTTP daemon stops and starts well below a second, the interruption of service is quite minimal. The overall point is that JADE provides safe and automated repair without hindering the legacy system performance, being fully compatible with clustered legacy systems tuned for high availability.

Finally, we experimented with the self-self-repair behavior of JADE itself and its overhead on the ability of JADE to repair managed legacy systems. We kept the failure of a Tomcat but forced a simultaneous failure of one of the JADE 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. Again, the repair of Tomcat and of JADE can be done without impacting the overall availability of the Web server.

6.2. Autonomic Protection

This section presents the experiment we made to evaluate our autonomic protection system. We first discuss the reactivity of our self-protection when an illegal communication is detected. We then evaluate the performance penalty induced by our self-protection system.

Our first experiment measures the time between the detection of an illegal communication and the isolation of compromised nodes. We have reproduced the scenario described in Figure 7, measuring the delay between the detection of an illegal communication coming for node 4 and the firewall reconfiguration on node 1, 2, and 9 in order to isolate node 5. The average time measured over 1000 runs is 2.133 ms with a 0.146 ms standard deviation. Hence, our prototype is very reactive and can quickly block an intruder.

Our second experiment measures the impact of protection on the performance of RUBiS. The deployed JEE architecture corresponds to that of Figure 7. The load injector of RUBiS emulates a variable number of clients, from 0 to 3000 in our experiments. The results are depicted in Figures 10 and 11. Figure 10 illustrates a read-only scenario (browse_only_matrix) whereas Figure 11 is about a read-write scenario (default_matrix). In both scenarios, we progressively increment the number of clients until we reach the saturation point. We compare the throughput with and without the self-protection system. The results show that for each matrix, the throughput with and without the self-protection system are very close (between 0 and 4% overhead).

One essential performance factor is the number of filtering rules in the firewall. Indeed, when a network packet goes through Netfilter, it is compared with each rule

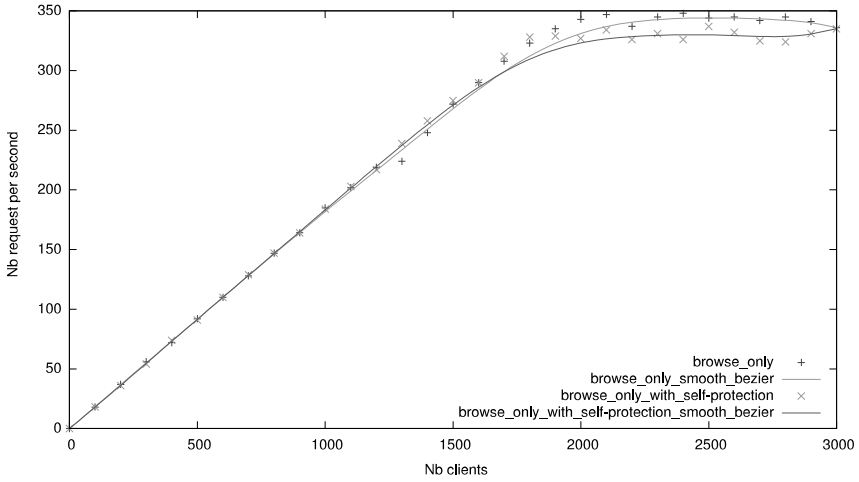

Fig. 10. Throughput for the browse_only_transitions matrix.

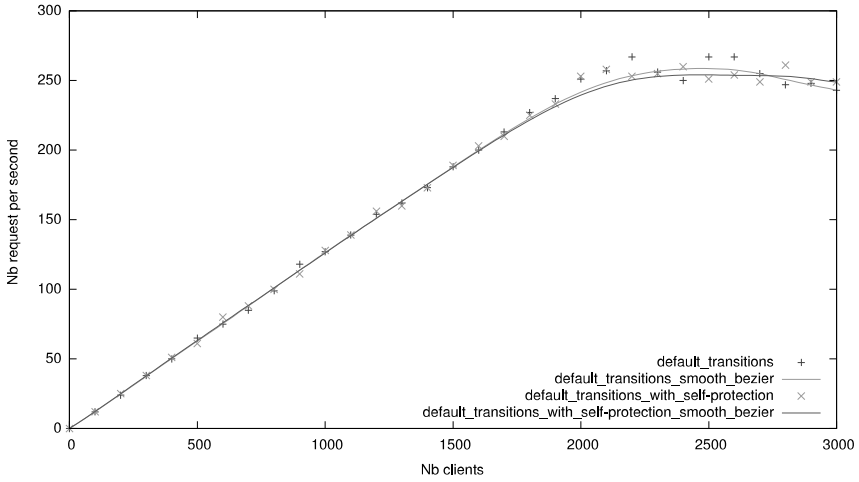

Fig. 11. Throughput for the default_transitions matrix.

following a priority until all rules are challenged or one of the rules matches (in the case of an illegal packet). Therefore the number of rules configured in firewalls impacts performance. In the previous results, we have about a dozen rules in each firewall. To check the scalability of our system, we have inserted 100 additional rules in each firewall before the 10 real rules that were generated for our JEE architecture. This number of rules in each machine represents a medium-size cluster composed of approximately 50 nodes. Results of this latter experiment are given in Figure 12. We can see that the overhead induced by the additional rules remains very low.

Results depicted in Table I represent the average response time with and without self-protection. We only evaluated the delay for read requests (browse_only_matrix) because they are the fastest and therefore the ones most penalized by the self-protection overhead. The given numbers are averaged over a thousand runs, with warmed-up server caches. The overhead is very low, with a 3.5% maximum.

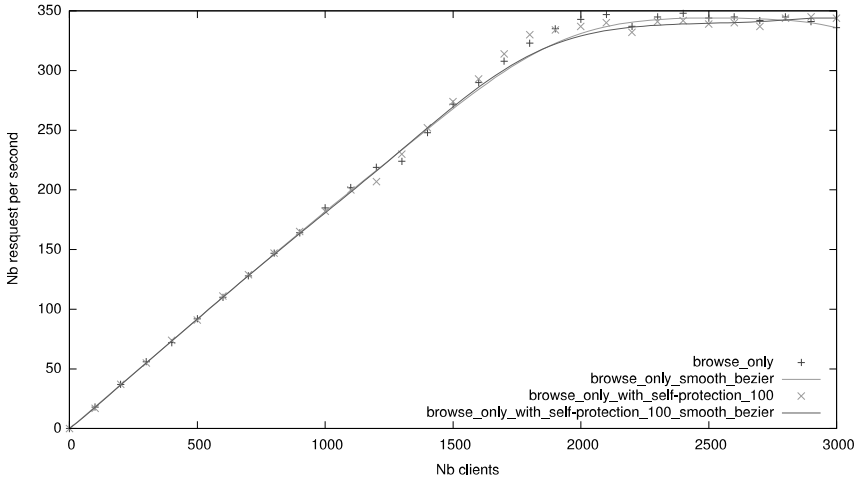

Fig. 12. Bandwidth for the browse_only_transitions matrix.

Table I. Response Time (Milliseconds)

Requests	Without self-protection	With self-protection
Home	2.374	2.380 (+0.25%)
Browse	2.354	2.378 (+1.02%)
BrowseCategories	3.985	4.069 (+1.10%)
SearchItemsInCategory	11.503	11.764 (+2.27%)
BrowseRegions	4.263	4.361 (+2.30%)
BrowseCategoriesInRegion	4.202	4.349 (+3.50%)
SearchItemsInRegion	19.960	20.240 (+1.4%)
ViewItem	3.352	3.361 (+0.26%)
ViewUserInfo	4.034	4.089 (+1.36%)
ViewBidHistory	7.474	7.728 (+3.40%)

7. RELATED WORK

Numerous works have focused on autonomic management of legacy systems. In Section 7.1, we consider autonomic management systems that deal with legacy applications. We then focus on architecture-based management frameworks that mostly rely on a formal or semiformal description of the managed system structure, typically expressed in terms of components and bindings using an architecture description language (ADL). In Section 7.2, we present architecture-based management frameworks based on nonreflective component models. In Section 7.3, we describe architecture-based management frameworkbased on reflective component models. Finally, Section 7.4 compares our work to frameworks encompassing a model@runtime approach.

7.1. Legacy Management Framework

Most autonomic management approaches for legacy systems are based on ad hoc solutions that are tied to a particular context. This reduces the reusability of the management services and policies; they 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 many projects provide ad hoc solutions for self-healing or self-optimization concerns.

For instance, Urgaonkar and Shenoy [2005], Appleby et al. [2001], and Norris et al. [2004] have considered the management of a dynamically extensible set of resources in the context of Internet services. Soundararajan et al. [2006] propose a self-optimized dynamic provisioning algorithm that specifically targets a cluster of databases. Pradhan et al. [2002] describe a solution to provide adaptation to changing workloads specifically for Web servers. In the same way, the JAGR project [Candea et al. 2003] provides a solution for self-recoverability in the context of Enterprise Java Beans running into the JBoss application server.

Other systems like KX [Parekh et al. 2006] 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 decisions) 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 JADE, wrapper components tightly coupled to the target system.

7.2. Framework Based on Nonreflective Component Models

Rainbow [Cheng et al. 2004; Dashofy et al. 2002], Darwin [Georgiadis et al. 2002] and Willow [Rowanhill et al. 2004] are good representatives of frameworks based on nonreflective component models.

Rainbow [Cheng et al. 2004] is an architecture-based management framework that supports self-adaptation of software systems. It uses an abstract architectural model to monitor the runtime properties of an executing system, evaluates the model for constraint violations, and if a problem occurs, performs global adaptations of the running system. One main objective of Rainbow is to favor 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 and the architecture layer, and vice versa.

In contrast, we only consider one layer of management in JADE, which is composed of reflective components representing the managed elements and providing, by reification, an architectural model of the runtime system. Moreover, JADE is inherently distributed, while the Rainbow framework is based on a centralized design. Finally, Rainbow concentrates on the system adaptation in terms of autonomic policies and event processing—it does not address software deployment and the self-self management challenges. In particular, Rainbow neither detects nor repairs its own failures. Furthermore Rainbow does not address its own protection against intrusions.

Dashofy et al. [2002] propose a framework for creating architecture-based autonomic systems focused on event-based software architectures that are suited for managed legacy systems that 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 language. The framework is composed of a specific component, called an architecture evolution manager, that can instantiate and update a running system whenever its architectural description changes. This component is therefore

responsible for managing the mapping between the running system and its architectural description. As in Cheng et al. [2004], this approach requires a mapping between the architectural description and the running system, which is automatic with JADE. Furthermore, the aspects related to the reliability of the components are presented as future work; self-self-management has not been taken into account.

Darwin [Georgiadis et al. 2002] proposes a component model based on an explicit architectural specification expressed in the alloy language [Jackson 1999]. 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-organizing 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 adjusts the component's configuration in accordance with the configuration view.

The component model proposed by Darwin more specifically targets self-organizing 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 large. While the self-organizing properties of this component model are interesting for obtaining autonomic capabilities, it could not be directly used as a basis for a management framework like JADE since it would imply providing a decentralized design of our autonomic managers in order to embed a copy of them within each component. Supporting such a design without involving a complex coordination between managers becomes a challenge since our autonomic managers often take global decisions concerning more than one individual component.

Finally, Willow [Rowanhill et al. 2004] also addresses the management of distributed applications through an architecture-based approach. Willow provides self-repairing and self-protection of very large-scale applications through a hierarchical management approach. It uses a relevant communication mechanism providing a dedicated event dissemination. The focus is on combining hierarchical and widely distributed management rather than on providing self-self behaviors.

7.3. Reflective Component Framework

With reflective component models, managed systems are implemented as a collection of interconnected components enhanced with a metalevel that provides introspection and reconfiguration capabilities on the component structure. The metalevel directly provides a causally connected representation of the component structure, mainly by ensuring that any change performed on the component structure at the metalevel are reported at the base level. Blair et al. [2004] consider the use of reflective middleware to develop self-managing systems as a challenging research direction. Our work on JADE falls in this category of systems projects such as OpenORB [Coulson et al. 2002], Plastik [Batista et al. 2005], and FORMAware [Moreira et al. 2002].

OpenORB [Coulson et al. 2002] is a middleware platform built around a well-founded reflective lightweight component model called OpenCOM. Like the fractal reflective component model used in JADE, the OpenCOM runtime provides support for a specializable and extensible metalevel model that provides 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 built-in support for building highly flexible distributed architectures

that ensure reconfiguration integrity. As we argue in JADE, the authors of OpenORB state that one needs a reflective component-based middleware to build an autonomic management system on top of it. However, they do not further investigate the necessary mechanisms and policies in such an autonomic system. In particular, they do not address the challenges of self-self-management in OpenORB.

The reflective OpenORB platform is used in the Plastik infrastructure [Batista et al. 2005], which follows an architecture-based management approach relying on the reified architecture provided by OpenORB. Plastik focuses 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 Cheng et al. [2002], conforms to a design pattern proposed by Rakic et al. [2002], which exposes architectural style requirements for building self-managed systems. Architectural styles could also be considered in JADE by more advanced autonomic managers, working on semantically higher-level reconfiguration operations.

The OpenORB platform is also used in the FORMAware project [Moreira et al. 2002]. FORMAware proposes extending architectural reflection for capturing domain-specific semantics and using it for safely governing architecture adaptations. As in Plastik, FORMAware uses the notion of architectural styles to impose explicit constraints for observing and managing the architecture. Their architectural styles define a set of formally specified constraints over an architecture, as well as how to carry out reconfigurations in terms of high-level architectural operators. A translation service maps the high-level architecture operations into lower-level systems operations acting on runtime components.

7.4. Models@runtime Approach

Models@runtime [Blair et al. 2009] is a new research trend that leverages model-driven engineering techniques (MDE) at design time as well as at run time. Like reflective frameworks, models@runtime promotes a causally connected representation of the underlying system. However such representation is based on the artifacts produced from the MDE process and the software engineering methodologies employed. Models@runtime focuses on the structure, behavior, and goals of the system from a problem space perspective while reflective approaches manipulate lower level abstractions that are related to the computation model.

Cheng et al. [2009] and Morin et al. [2009] are examples of models@runtime approaches. Cheng et al. [2009] is based on an architectural model to support the design of component-based adaptive systems. The system's adaptation logic is specified using a state machine, where each state represents a particular configuration. Each transition describes, (1) when the system must switch from one configuration to another, and (2) what reconfiguration script must be used to update the system. Cheng et al. [2009] use this model to generate configuration files and ECA adaptation policies that can be dynamically inserted at runtime.

Morin et al. [2009] follow a models@runtime approach for specifying and executing dynamically adaptive software systems. This proposal aims at reducing the complexity of such systems by using both model-driven and aspect-oriented techniques. Four metamodels are used to explicit the system's variability (using a feature diagram), the environment, the adaptation logic, and the system architecture. Aspect-oriented techniques are used to generate architectures by weaving aspects associated with features, instead of describing all the possible configurations. Once these configurations are verified, model-driven techniques are used to produce the reconfiguration scripts

that allow the system to switch from the current configuration to a target configuration depending on the runtime context.

Since JADE is built as a reflective system, its metamodel and its reconfiguration capabilities are based on the computational model incarnated by Fractal. It would be interesting to study how JADE could be extended towards a models@runtime approach in order to take into account design time aspects when reconfiguring a system.

While the models@runtime approach appears promising, it is still in its infancy, with very few real use cases. It is difficult to evaluate the impact of the approach on system performance. This is potentially the case when considering the overhead of maintaining the necessary causal relationship between the model and the running system in distributed systems with nodes and network failures.

8. CONCLUSION AND LESSONS LEARNED

During the five years of the JADE project, we have learned several important lessons that we wished to share. Some, but not all, confirm our design and technical choices. Also, we would like to share some insight on the impact of autonomic management on the future of distributed systems, as well as sketch the main open issues as we see them.

We feel that the cornerstone of our approach, the use of a reflective component model, was essential in reaching full autonomic behavior. The importance of having a model for the definition of a minimal set of management operations cannot be understated. Components made wrapping easier and less error-prone and supported the concept of an architecture-based approach. Additionally, a component-oriented approach made our recursive design quite natural.

In our quest for reaching fully autonomic behaviors, it seems that not all managers have equal requirements. Some autonomic behaviors seem to inherently require a self-self approach, while others can perfectly fulfill their role with a simpler self approach. For instance, we discussed in detail the challenge of autonomic repair and autonomic protection in this article. These two autonomic behaviors must be replicated and must self-apply. Anything less, and it is difficult to declare an autonomic property. However, once protection and repair behavior are available, many other managers may rely on being protected and repaired. Our performance managers, such as load balancers or quality-of-service monitors, are typical examples of managers that do not require any self-self behavior.

On a different topic, our failure assumptions were often questioned, as well as our assumptions on the execution model for the legacy systems we can manage with JADE. In all practical aspects, we feel that our fail-stop assumption was acceptable and allowed us to focus on reaching autonomic behavior. It seems important however that this work be extended to take into account Byzantine failures. We feel that many of the existing solutions would apply in our context, but much more research work is needed.

Regarding our assumptions around a loose coupling of legacy systems, they have been quite confirmed through our practical involvement with real systems. The key point is that loose coupling is mandatory for allowing partial failures. The more coupled are the legacy systems, the less likely some parts of the overall system will resist any partial failure. In contrast, loose coupling provides the basis for resisting the spreading of failures and thereby offers the opportunity for incremental repair. Furthermore, we feel that trying to do autonomically what administrators do manually, using the same administration capabilities, was very productive in focusing our work on providing concrete and understandable autonomic behaviors.

As a corollary to loose coupling, our experience suggests that some distributed systems still have to mature. We detailed some cases in this article. For instance, the HTTP daemon's inability to reread its configuration without any interruption of

service. This clearly illustrates that the design of subsystems has not yet integrated the possible presence of autonomic behaviors. We are still very much in the era of simple watchdogs, repairing standalone systems. We feel that core autonomic behaviors such as repair and protection need to find their way in replacing operating-level services such as the InetDaemon.

A similar statement can be made about deployment. While deploying Java wrappers and components was easy, the deployment of legacy systems is complex and extremely platform-specific. The recent advance in virtualization technologies seems to suggest that generic deployment solutions are foreseeable in the near future. This evolution will certainly help the establishment of autonomic management solutions, as deployment underlies so many autonomic behaviors.

On a different line of thought, it is our experience that architecture-based autonomic behaviors are inherently challenged by highly dynamic systems. In JADE, we support the management of systems that we call *admin-dynamic* systems. These systems evolve dynamically, but for administration reasons (protection, maintenance, repair, or even load balancing). This means that architectural changes are expressed on the architecture and then applied onto the managed system. In contrast, much research work is needed for the autonomic management of systems where architectural changes come from the managed system itself. As an extreme example, peer-to-peer systems are challenging not only because of their high churn rate in terms of nodes but also in terms of the dynamicity of the bindings between these nodes.

Finally, scalability is challenging for autonomic behaviors. As the size of the managed system grows, several facets of our design need to be reevaluated. It is clear that a failure detector would need an adequate design for a large scale network. With larger scale networks, network partitioning must be considered carefully. Similarly, a single description of the architecture of the managed system does not scale. This would suggest adapting our current approach to apply it at the granularity of single administration domains and to work out coordination among autonomic behaviors across domains.

REFERENCES

AMZA, C., CECCHET, E., CHANDA, A., COX, A., ELNIKETY, S., GIL, R., MARGUERITE, J., RAJAMANI, K., AND ZWAENEPOEL, W. 2002. Specification and implementation of dynamic Web site benchmarks. In *Proceedings of the IEEE 5th Annual Workshop on Workload Characterization (WWC-5)*.

APACHE. *HTTP Server Project*. http://httpd.apache.org/.

APPLEBY, K., FAKHOURI, S., FONG, L., GOLDSZMIDT, G., AND KALANTAR, M. 2001. Oceano - SLA based management of a computing utility. In *Proceedings of the 7th IEEE International Symposium on Integrated Network Management*.

BATISTA, T., JOOLIA, A., AND COULSON, G. 2005. Managing dynamic reconfiguration in component-based systems. In *Proceedings of the European Workshop on Software Architecture*.

BLAIR, G., BENCOMO, N., AND FRANCE, R. 2009. Models@Run.Time. *Computer 42,* 10, 22–27.

BLAIR, G. S., COULSON, G., AND GRACE, P. 2004. Research directions in reflective middleware: the Lancaster experience. In *Proceedings of the 3rd Workshop on Adaptive and Reflective Middleware (ARM)*.

BRUNETON, E., COUPAYE, T., LECLERCQ, M., QUÉMA, V., AND STEFANI, J. 2006. The fractal component model and its support in Java. *Softw. Practice Exper*. (Special Issue on Experiences with Auto-Adaptive and Reconfigurable System) *36,* 11–12, 1257–1284.

CANDEA, G., KICIMAN, E., ZHANG, S., KEYANI, P., AND FOX, A. 2003. JAGR: An autonomous self-recovering application server. In *Proceedings of the 5th International Workshop on Active Middleware Services (AMS)*.

CECCHET, E., MARGUERITE, J., AND ZWAENEPOEL, W. 2004. C-JDBC: Flexible database clustering middleware. In *Proceedings of the USENIX Annual Technology Conference, Freenix Track*.

CHENG, B. H. C., SAWYER, P., BENCOMO, N., AND WHITTLE, J. 2009. A goal-based modeling approach to develop requirements of an adaptive system with environmental uncertainty. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MoDELS)*. 468–483.

CHENG, S., GARLAN, D., SCHMERL, B., SOUSA, J., SPITZNAGEL, B., AND STEENKISTE, P. 2002. Using architectural style as a basis for self-repair. In *Proceedings of the 3rd Working IEEE/IFIP Conference on Software Architecture*.

CHENG, S. W., HUANG, A. C., GARLAN, D., SCHMERL, B., AND STEENKISTE, P. 2004. Rainbow: Architecture-based self adaptation with reusable infrastructure. *IEEE Computer 37,* 10, 46–54.

COULSON, G., BLAIR, G. S., CLARKE, M., AND PARLAVANTZAS, N. 2002. The design of a configurable and reconfigurable middleware platform. *Distrib. Comput. 15,* 2.

DASHOFY, E., VAN DER HOEK, A., AND TAYLOR, R. 2002. Towards architecture-based self-healing systems. In *Proceedings of the 1st ACM Workshop on Self-Healing Systems*.

GEORGIADIS, I., MAGEE, J., AND KRAMER, J. 2002. Self-organizing software architectures for distributed systems. In *Proceedings of the 1st Workshop on Self-Healing Systems*.

GRAY, J. 1986. Why do computers stop and what can be done about it? In *Proceedings of the Symposium on Reliability in Distributed Software and Database Systems*.

GRAY, J. 1990. A census of tandem system availability between 1985 and 1990. Tech. rep., Tandem Computers.

GUERRAOUI, R. AND SCHIPER, A. 1996. Fault-tolerance by replication in distributed systems. In *Proceedings of the International Conference on Reliable Software Technologies*. Springer Verlag.

JACKSON, D. 1999. Alloy: A lightweight object modelling notation. *MIT Lab for Computer Science*.

KALYANAKRISHNAM, M., KALBARCZYK, Z., AND IYER, R. 1999. Failure data analysis of a LAN of Windows NT based computers. In *Proceedings of the 18th IEEE Symposium on Reliable Distributed Systems*.

KEPHART, J. AND CHESS, D. 2003. The vision of autonomic computing. *IEEE Computer 36,* 1.

MOREIRA, R., BLAIR, G., AND CARRAPATOSO, E. 2002. FORMAware: Framework of reflective components for managing architecture adaptation. In *Proceedings of the 3rd International Workshop on Software Engineering and Middleware*.

MORIN, B., BARAIS, O., NAIN, G., AND JEZEQUEL, J.-M. 2009. Taming dynamically adaptive systems using models and aspects. In *Proceedings of the IEEE 31st International Conference on Software Engineering (ICSE)*. IEEE Computer Society, Los Alamitos, CA, 122–132.

NETFILTER. Firewalling, NAT, and packet mangling under Linux. http://www.nefilter.org.

NORRIS, J., COLEMAN, K., FOX, A., AND CANDEA, G. 2004. OnCall: Defeating spikes with a free-market application cluster. In *Proceedings of the 1st International Conference on Autonomic Computing (ICAC)*.

OPPENHEIMER, D., GANAPATHI, A., AND PATTERSON, D. 2003. Why do Internet services fail, and what can be done about it? In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems*.

PAREKH, J. J., KAISER, G. E., GROSS, P., AND VALETTO, G. 2006. Retrofitting autonomic capabilities onto legacy systems. *Cluster Comput. 9,* 2, 141–159.

PRADHAN, P., TEWARI, R., SAHU, S., CHANDRA, A., AND SHENOY, P. 2002. An observation-based approach towards self-managing Web servers. In *Proceedings of the 10th IEEE International Workshop on Quality of Service*.

RAKIC, M., MEHTA, N., AND MEDVIDOVIC, N. 2002. Architectural style requirements for self-healing systems. In *Proceedings of the 1st Workshop on Self-Healing Systems*.

ROWANHILL, J. C. , VARNER, P. E., AND KNIGHT, J. C. 2004. Efficient hierarchic management for reconfiguration of networked information systems. In *Proceedings of the Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*.

SICARD, S., BOYER, F., AND DE PALMA, N. 2008. Using components for architecture-based management: The self-repair case. In *Proceedings of the 30th International Conference on Software Engineering*.

SOUNDARARAJAN, G., AMZA, C., AND GOEL, A. 2006. Database replication policies for dynamic content applications. In *Proceedings of the 1st EuroSys Conference*.

URGAONKAR, B. AND SHENOY, P. J. 2005. Cataclysm: Policing extreme overloads in Internet applications. In *Proceedings of the 14th International Conference on World Wide Web (WWW)*.

Received March 2009; revised September 2009, February 2010; accepted July 2010