SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2010; **00**:1–20
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe

Full Autonomic Repair for Distributed Applications

FABIENNE BOYER[1],
NOEL DE PALMA[1], OLIVIER GRUBER[1] and SYLVAIN SICARD[1]

[1] *LIG / University of Grenoble - France*

SUMMARY

Grid or cloud environments leverages the need for self-repair solutions that resist and repair their own failures, something not yet ensured by existing solutions. In this paper, we describe the JADE Autonomic Repair System for legacy applications deployed in a grid or cloud environment. JADE is based on three main design principles. First, legacy applications are wrapped with Java objects, obtaining a uniform set of management operations over the heterogeneous legacy management capabilities. Second, to gain full autonomy, we adopt a replicated design combined with a recursive approach that makes JADE appear to JADE as any distributed application it manages and repairs. Finally, to scale, we rely on tiling the distributed environment and structuring our repair system per tile. To our knowledge, our repair system is the only one that is designed to scale and is fully autonomic, repairing not only the failures of the managed system but also its own. Our repair system has been tested in various realistic scenarii. Copyright © 2010 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: autonomic computing, repair management, distributed systems, component models

1. INTRODUCTION

Computational grids or clouds are expected to become a common way to deploy and execute distributed applications in the next future, which suggests autonomic management capabilities [25]. During the last few years, autonomic computing has been an active research topic, pursuing the goal of providing self-deployment, self-repair, self-optimization and self-protection facilities as initially identified by Kephart [23, 22]. While much progress has been made, most autonomic services are not yet fully autonomic, still requiring the presence of a human administrator to deploy and manage them during their execution. In particular, existing self-repair services repair failures occurring in the applications they observe but do not survive their proper failures over time without human intervention [29][8][17][3].

We believe this partial autonomy to be a significant drawback of self-repair services, as it implies the presence of human administrators to observe the self-repair services running in the grid or cloud environment and repair them manually when they fail, generating potentially long average Mean-Time-To-Repair (MTTRs). Moreover, for scalability reasons, there may be a large number of self-repair services running in the environment and these services may not be independent the one with each other, forming a complex distributed application on their own. This raises an increased probability for human errors, as it is recognized that manual repairs are increasingly complex and error-prone [21][26]. At the end, such situation may lead to erroneous self-repair behaviors. While completely avoiding the presence of human administrators for supervising grid or cloud

*Correspondence to: E-mail: firstname.lastname@imag.fr

Copyright © 2010 John Wiley & Sons, Ltd.

environments would be unrealistic, a strong challenge for such environments is to limit manual repairs to exceptional situations only. The contribution of this paper is a fully autonomic self-repair service called JADE that solves the following three challenges:

- Fully autonomic repair: JADE not only repairs the distributed applications it manages but also repairs itself, a property that we call *self²-repair*. This is achieved by combining a recursive and replicated design. Through a recursive design, JADE appears to itself as a distributed system that it can manage and repair. By replicating JADE, each replica of JADE can watch over the other replicas and repair them if they fail.

- Scale: as the complexity and size of the set of managed distributed applications grows, JADE scales by tiling itself. A typical and natural tiling consists in using the different administrative domains of the underlying network topology. Each tile is locally managed by a complete JADE with limited and well-understood interactions across tiles.

- Unmodified legacy: JADE does not require to modify the applications it manages. It is able to repair these applications using the very same management capabilities that a human administrator would use, through wrapping their specific legacy management interface within a small and uniform set of management operations.

To achieve this, a core design principle adopted in JADE is to maintain at runtime the knowledge of the software architecture of the managed distributed system, which includes the description of th local configuration of subsystems and their interconnections. This knowledge, captured in the *System Image*, is used to analyze detected failures and plan the repair of lost subsystems, ensuring that failed subsystems are not only repaired but also correctly reinserted in the overall distributed system.

JADE is intended to repair distributed systems, composed on dozens of subsystems, that are deployed in small or large environments—such as clusters, grids, or clouds environments. Typical examples of such distributed systems are Web application servers, JEE servers, JMS applications, or more classical distributed systems such as replicated NFS servers. In the current state of our work, we only support fail-stop nodes and applications, leaving Byzantine failures as future work.

Our solution has been tested and validated through several usecases. We considered repairing Web application servers [10], for both the basic multi-tiered architecture and the clustered architecture. We also used our solution to repair JMS servers and reconnecting them to clients [15]. Finally, we experienced our solution with repairing a storage service based on Network File Systems (NFS). These experiences show that our approach is entirely compatible with high availability and delivers safe and fast recovery compared to a human administrator.

Compared to our previously published work in the context of the Jade autonomic management system [30] [9], this paper details the distributed protocol we designed to gain full autonomy and explains how we introduced tiles to gain scalability. It is structured as follows. In Section 2, we describe the overall distributed structure of JADE. In Section 3, we present how JADE self-repairs distributed applications. In Section 4, we present how JADE self²-repairs. In Section 5, we report on two concrete usecases of JADE, repairing Web application servers and repairing a storage service. In Section 7, we discuss related works. In Section 8, we conclude.

2. OVERALL DISTRIBUTED REPAIR STRUCTURE

The overall distributed structure of our repair service is organized around two main layers: the *Wrapped Distributed Environment* and the *System Image*. Section 2.1 describes the Wrapped Distributed Environment that wraps the heterogeneous legacy subsystems, making them manageable through a small and uniform set of management operations. Section 2.2 presents the *System Image* that captures the software architecture of the managed distributed system. Finally, section 2.3 presents how these two layers are partitioned such as forming areas called *Tiles* that can be observed and repaired by separate repair service instances.

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

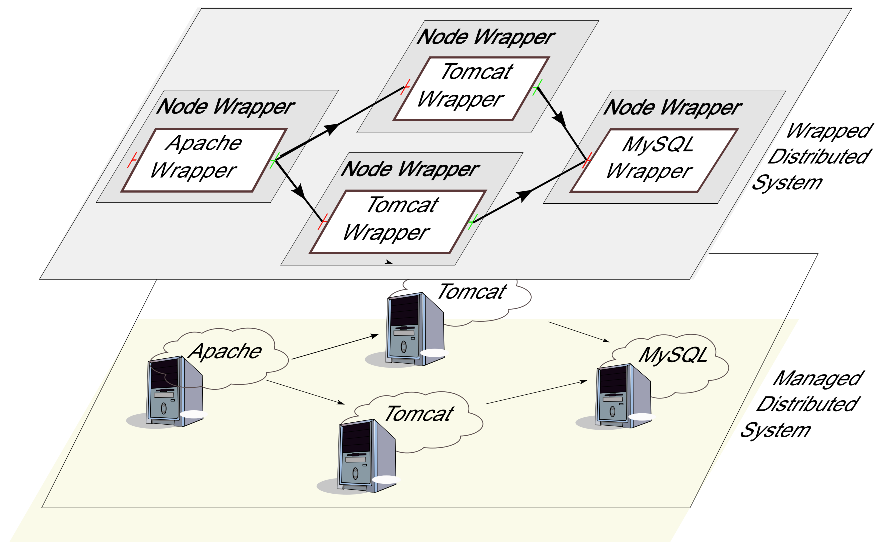

Figure 1. Managed Distributed Environment

2.1. *The Wrapped Distributed Environment*

In JADE, wrapping a distributed environment hosting legacy subsystems is about wrapping individual subsystems as well as the nodes hosting them (real or virtual machines). Figure 1 depicts the wrapping of a typical clustered JEE application server.

Each wrapper is implemented as a Java object that is always co-located with the subsystem it wraps. Wrappers implement a set of *uniform management operations* [30] that provide control over the lifecycle, the attributes, and the bindings of a wrapped subsystem. The lifecycle is about starting and stopping a subsystem. The implementation of these methods is often reduced to manipulating configuration files or launching already existing scripts. For instance, wrapping the Apache HTTP daemon relies on the existing *start* and *stop* scripts. The attributes capture, as key-value pairs, the configuration data of the wrapped subsystem, usually found in configuration files. In other words, subsystems are configured through setting attribute values. For instance, an attribute may capture the TCP/IP port used by an Apache daemon to listen to incoming HTTP requests.

Bind and unbind management operations allow to manage configuration dependencies (e.g., interconnections) between subsystems. The most common case of such dependencies is the presence of communication channels between wrapped subsystems. For instance, the connection from an Apache HTTP daemon to a Tomcat Servlet engine would be represented by a binding from the HTTPD wrapper to the Tomcat wrapper. The binding enables the HTTPD wrapper to invoke remote methods on the Tomcat wrapper and thereby requests the necessary configuration information about the connection that needs to be created. Having that information, the HTTPD wrapper can then update the local configuration file of its wrapped Apache HTTP daemon.

This example illustrates an important point. Wrappers do not establish communication channels between the subsystems they wrap; wrappers apply management operations on their wrapped subsystems so that these subsystems establish the correct communication channels themselves. Therefore, wrappers and bindings are not involved in the actual communication between subsystems—wrapped legacy subsystems communicate directly as they would have without being wrapped.

When a management operation is invoked on a wrapper, it is not supposed to be synchronously (i.e., immediately) executed onto the wrapped legacy subsystem. Indeed, a crucial requirement

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

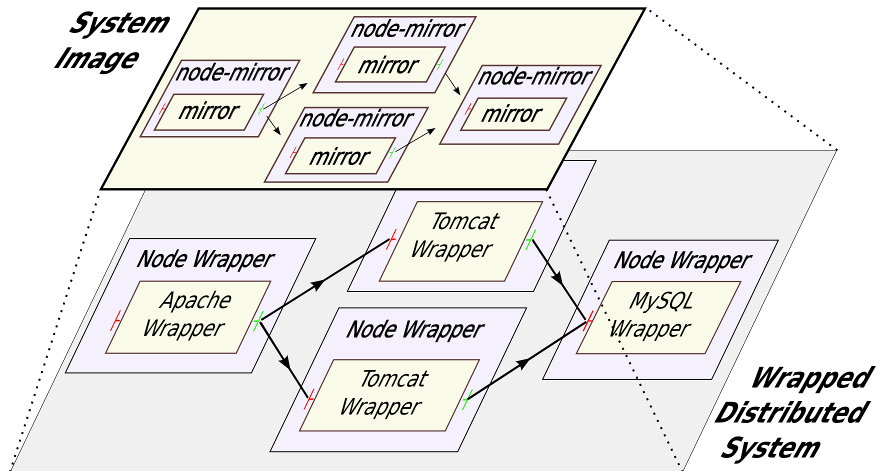

Figure 2. System Image

about wrappers, that we realized experimentally, is to offer each wrapper enough freedom about how to apply management operations onto its wrapped subsystem. Some subsystems must be configured before they are started while others must be configured once started. Some subsystems must be restarted in order to take into account a configuration change, others can respond to signals or periodically check their configuration. This freedom requirement is usually not taken into account by usual wrapping techniques [31]. To address this, we introduced the concept of an *apply session* that groups management operations. Within each apply session, bracketed by a *beginApply* and *endApply* invocations, each wrapper may reorder as it sees fit the management operations it receives. However, each wrapper must apply all received management operations before it replies to the *endApply* invocation.

Our Apache-Tomcat example illustrates this situation. When a node running Tomcat fails, JADE will relocate the Tomcat subsystem on a new node. To chose that new node, JADE has specific knowledge that captures the compatibility between subsystems and nodes. It will then notify the Apache wrapper of this relocation, through a pair of *unbind-bind* operations. The *unbind* operation on the Apache wrapper notifies that wrapper that the Tomcat it was connected to has failed. The *bind* operation notifies the wrapper of the restart of the Tomcat server. This operation provides the remote reference of the new wrapper for the relocated Tomcat. Using that remote reference, the Apache wrapper can inquire about new configuration attributes, if any, updating the configuration file of its wrapped Apache HTTP daemon. Finally, the Apache wrapper will stop and restart the HTTP daemon, re-establishing the communication channel with the relocated Tomcat.

To relocate subsystems, JADE relies on the wrapping of nodes. Node wrappers provide JADE with the ability to deploy subsystems and create their wrappers on the node they wrap. Thus node wrappers provide both a description of the execution environment (what are the machines composing the environment) and a description of the software subsystems that are running in this environment. At initialization time, node wrappers are created from a static description of the environment, where each node is mainly described by its characteristics, its IP address and a logical name that is given by the administrator.

2.2. *System Image*

The System Image captures the necessary knowledge of the managed distributed system. The challenges are the following. First, we need to capture this knowledge in a way that supports easy introspection and reconfiguration of the managed distributed system. Second, this knowledge has to survive the failure of managed subsystems.

Copyright © 2010 John Wiley & Sons, Ltd.

Prepared using speauth.cls
DOI: 10.1002/spe

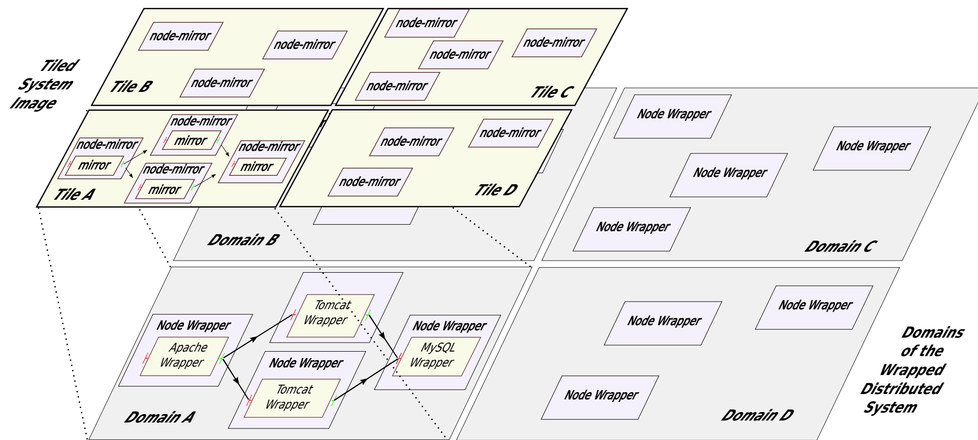

Figure 3. Tiled Distributed Structure

As depicted in Figure 2, the System Image is composed of *mirrors*, where a mirror is a plain Java object that provides a reflection of the wrapper it mirrors. This reflection includes the lifecycle status (started or stopped), the attributes (configuration key-value pairs), and the bindings. The bindings between mirrors are isomorphic to bindings between wrappers. In other words, if there is a binding between two wrappers, there is an equivalent binding between their mirrors. Additionally, the mirror of a node wrapper, called a *node-mirror*, captures isomorphically the knowledge of the wrappers deployed on that virtual or hardware node. In other words, if a node-mirror contains a mirror, the corresponding node wrapper contains the wrapper of that mirror.

Thus mirrors correspond to remote copies of wrappers. By introspecting mirrors in the System Image, our Repair Manager knows which wrappers are deployed where, if they are started or not, what are their attributes, and what are the dependencies that link them. This introspection using the System Image is central to our failure analysis since mirrors survive failures while wrappers do not. Indeed, when a node fails, both the wrappers and the subsystems they wrap are lost. Therefore, mirrors are the memory of what was deployed on failed nodes and how it was configured.

By reconfiguring mirrors, the Repair Manager indicate how the subsystems lost to a failure must be recreated, potentially relocated, and reinserted in the running distributed system. By reconfiguring, we mean that the Repair Manager may create or delete mirrors, change attributes, as well as create, delete, or change bindings between mirrors. This reconfiguration appears atomic to the managed distributed system, meaning that wrappers and their wrapped subsystems remain unaffected, the reconfiguration solely happens on mirrors at first. Then, as detailed in Section 3, at commit time, the reconfiguration is applied and the necessary management operations are forwarded onto concerned wrappers.

2.3. *Overall Tiled Distributed Structure*

To scale, our overall distributed structure is tiled as depicted in Figure 3. Per tile, we have a complete JADE system composed of a System Image, a Repair Manager, and a Failure Detector.

Each tile is also composed of a certain number of managed nodes, each managed node belonging to one and only one tile. A likely tiling follows existing administrative domains of the underlying network topology, using one or more tile per domain. In other words, each tile captures the architecture of some of the subsystems deployed in a network domain. Consequently, each System Image only mirrors local subsystems belonging to its tile.

It is important to note that bindings between mirrors may be either local to a tile or remote between tiles. This is entirely transparent to our repair process since our mirrors are regular Java objects that can be distributed transparently. Hence, the repair process introspects and reconfigures mirrors regardless of their tiling. However, by design, a repair process is mostly local to a tile, as explained in the next section.

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

3. THE REPAIR PROCESS

The repair process in JADE happens per tile, in a completely asynchronous and decentralized manner. Each tile watches over its wrapped subsystems and detects their failure. We consider fail-stop subsystems and nodes, Byzantine failures being future research. When a failure is detected in a tile, the Repair Manager of that tile is informed of that failure and it starts a session composed of three phases: an analysis, a reconfiguration, and a commit.

The purpose of the analysis is to determine what has been lost to the detected failure. This is done by introspecting the mirrors of the tile where the failure occurred, building the set of *impacted mirrors*, that is, the set of mirrors whose corresponding subsystems have been lost to the reported failures.

Once the impacted mirrors are known, the repair session can compute the set of *impacted tiles*. The tile where the failure occurred, called the current tile, is obviously in that set. But subsystems in other tiles may have bindings to one or more lost subsystems in the current tile. Thus, the repair session on the current tile has to be extended to the impacted tiles, ensuring that there is one and only repair session in progress across the impacted tiles. If another repair session is already in progress in one of the impacted tiles, the repair process in the current tile will simply wait before proceeding. In case of the even more unlikely event of a deadlock situation in extending repair sessions, we simply abort one session (the analysis is a read-only step, so the abort is trivial) and we retry later.

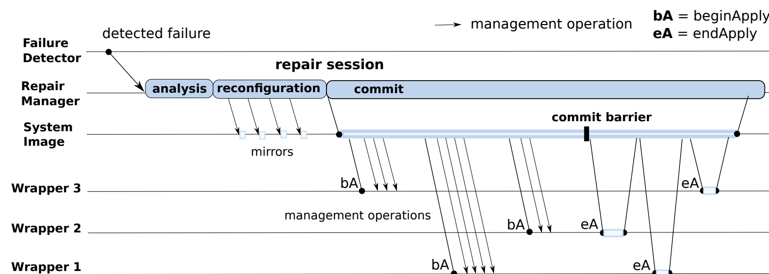

Figure 4. Repair Session

Once the repair session over impacted tiles has been established, the repair session may proceed to the reconfiguration phase, as depicted in Figure 4. Knowing the impacted mirrors, it reconfigures them in the System Image in three main steps. The first step manages the replacement of the lost subsystems. For each impacted mirror, the repair session clones that mirror, adding the clone to a node mirror, thereby expressing where the failed subsystem will be relocated and restarted at commit time. If the failure was due to a node failure, all the subsystems that were running on the failed node are relocated on a new node. Otherwise, a local restart is preferred. It is important to point out that the clone operation copies the attributes and the bindings. The second step repairs stale bindings: bindings to impacted mirrors. For each stale binding to an impacted mirror, it is removed and replaced by a new binding to the clone of the impacted mirror. Finally, the last step restarts the repaired subsystems and disposes of impacted mirrors. These three steps are summarized in the algorithm given in Listing 1.

Once the System Image has been reconfigured, repairing the detected failure, the final phase of the repair session is to commit—bringing the managed subsystems back in sync with the already reconfigured System Image. During this commit phase, wrappers of impacted mirrors (failed subsystems) will be disposed of, a step during which disposed wrappers clean up their local configuration (such as removing lock files or temporary files). Then, a new wrapper will be created for any clone mirror that the repair session created in the System Image. Finally, for any *unbind* operation that the repair session did in the System Image, a corresponding *unbind* operation is invoked on the concerned wrappers. When receiving such *unbind* operation, a wrapper may request its wrapped legacy to close stale communication channels. Symmetrically, for any *bind* operation the the repair session did in the System Image, a corresponding *bind* operation is invoked on the concerned wrappers. When receiving such operation, a wrapper usually updates the configuration

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

Listing 1. Reconfiguration phase of the repair session

```
for any failed subsystem S, reconfigure the System Image as follow:

   let M be the mirror associated with S
   let N be the node hosting S

   // repair through relocation / local restart
   if failed(N)
      let N' be the relocation node for N
   else N' = N

   // replace the failed subsystem
   remove M from N and dispose M
   create a new mirror M' configured as M
   add M' in N'

   // repair stale bindings
   for any mirror X bound to M
       X.unbind(M)
       X.bind(M')

  // start the repaired subsystem
   start M'
```

of its wrapped legacy subsystem, thereby requesting that the legacy subsystem recreates correct communication channels to the newly restarted subsystems.

Per wrapper, all invoked management operations are bracketed by a *beginApply* and an *endApply*, as discussed in Section 2.1. Wrappers that are disposed of are treated first. Then across the remaining wrappers, JADE globally orders the *endApply* invocations according to the binding dependencies in the System Image. By default, a wrapper of a client subsystem is always processed after the wrappers of the server subsystems it is connected with. If bindings form a cycle, the administrator is requested to declare startup priority in the System Image. Globally ordering *endApply* invocations is important to ensures that a dependent wrapper knows that the wrappers it depends on have all finished their *endApply* invocation before it receives its own *endApply* invocation.

By the time all *endApply* invocations on wrappers have returned, the current repair session is finished. A new repair session may be opened to handle pending reports of detected failures. Once a repair session is opened, incoming reports of newly detected failures are queued and will only be taken into account in a follow-up repair session. A major design point is that once a repair session enters the commit phase, it always proceeds to completion, even though wrappers may fail when applying requested management operations. The reason is that any such failure occurring during a commit is treated as a failure that will be repaired in a follow-up session. From the perspective of the applications running in the overall distributed system, there is indeed no telling the difference between a subsystem that fails while applying a management operation that is part of a commit and a subsystem that fails right after that commit. Through such design, JADE not only manages failures occurring at execution time but also considers failures occurring at repair time, restarting the repair process for subsystems that fail while being repaired.

4. THE SELF^2 REPAIR PROCESS

To gain the self^2-repair property, one must propose a solution that tolerates and repairs failures of the JADE repair system—a distributed system of its own. We chose to replicate the JADE subsystems—namely the System Image, the Repair Manager, and the Failure Detector—using an active replication scheme such that a fault-tolerant repair service can be provided. However, failed JADE subsystems shall be repaired and reinserted over time without human intervention. We therefore designed JADE so that JADE is able to self^2-repairs using the very same approach that

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

it uses to repair any managed subsystem. This section discusses how we achieve this design that combines both recursion and replication. It describes the several challenges we faced even though an active replication scheme is a well-known technique.

4.1. Multigathercast references

Our first challenge is that most replication techniques apply to replicated servers that offer leaf operations. By leaf operations, we mean that each replica is the end of the process chain: the request is processed entirely locally at each replica. In contrast, we mix replicated and non-replicated subsystems in JADE. Indeed, we replicate the JADE subsystems—System Image, the Node Failure Detector, and the Repair Manager—but wrappers are not replicated. Since we use the object-oriented paradigm to design JADE, we have to design a protocol for remote object references that is capable of mixing a multicast and a gathercast semantics, which we call a *multigathercast* semantics. Multigathercast references, as depicted in Figure 5, support remote method invocations going from unreplicated objects (resp. replicated) to replicated ones (resp. unreplicated).

A method invocation to a replicated server subsystem requires a *multicast* semantics that is totally-ordered and reliable. Essentially, this translates in: (i) all correct skeletons receive method invocations in the same order, (ii) all correct skeletons receive a method invocation or none of them receives it. Furthermore, individual replicas may fail without blocking the reliable and totally-ordered multicast. Our implementation relies on JGroups[†], a toolkit that provides group membership services and a reliable totally-ordered message delivery to dynamic groups. We associate one multicast group identified by a Globally Unique IDentifier (GUID) per replicated JADE subsystem.

Symmetrically, a method invocation from a replicated client system requires a *gathercast* semantics to avoid duplicate execution of invoked methods. The gathercast semantics ensures that any method invocation requested by the client-side replicas is executed *at most once at each server-side replica*. To that end, each server replica maintains a cache of computed results, called the *invocationResultCache*. The corresponding algorithms are given in listing 2 and listing 3.

Listing 2: Client-side Multigathercast

```
Channel channel = ... ; // JGroups channel
Map pendingCalls =    ;  // result cache

methodCall(String signature, Object[] args):
   String callid = getNextCallid();
   pendingCalls.put(callId, null);
   channel.send(marshall(signature, args));
   callId.wait();
   return(pendingCalls.get(callId));

receiveResult(Message m):
   String callId = m.callid();
   // store the first reply received
   pendingCalls.put(callid, msg.result());
   callId.notify();
```

Listing 3: Server-side Multigathercast

```
Channel channel =     ; // JGroups channel
Cache invocationResultCache =     // result cache

handleMethodCall(Message m):
   if ((res = invocationResultCache.get(m.callid)) == null)
       res = invoke(m.signature, m.args);
       invocationResultCache.put(m.callid, res);
   channel.send(marshall(m.callid, res));
```

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

Copyright © 2010 John Wiley & Sons, Ltd.
Prepared using speauth.cls

Duplicate invocation messages are detected with a simple scheme based on a sequence number and client GUID. This works because replicated JADE subsystems are deterministic, each replica will invoke the same methods in the same order, generating the very same sequence numbers, without global synchronization.

Duplicate responses also need to be absorbed. Each client replica uses the first response it receives to unblock the caller and discards all future responses tagged with the same client GUID and the same sequence number. To support this, each stub maintains a cache of pending requests, called the *pendingCallsCache*, that remembers if a request is pending or if a result has been received already.

The introduction of two caches requires a cleanup protocol. In the absence of failures, the cleanup protocol is fairly easy since the replication cardinality is known for both the client and server sides. However, subsystem replicas may fail; we therefore extend our cleanup protocol using the fact that failed replicas are automatically repaired and reinserted by JADE.

Once failed replicas are repaired, the cleanup protocol goes back in normal mode where client and server replicas receive again all expected duplicate messages. Therefore, each replica can now autonomously maintain two thresholds on sequence numbers. The first threshold, the *invocationResultCache threshold*, is the highest sequence number for which the replica has seen all duplicates of an invocation message. Any entry in this cache that has a lower sequence number than the threshold can be safely removed. The second threshold, the *pendingCallsCache threshold*, is the highest sequence number for which the replica has seen all duplicates of a result message. Any entries in the pendingCallsCache that has a lower sequence number than this threshold can be safely removed. This simple protocol works because replicas are repaired and reinserted on the one hand and on the other hand correct replicas will receive all sent messages since we use a reliable multicast.

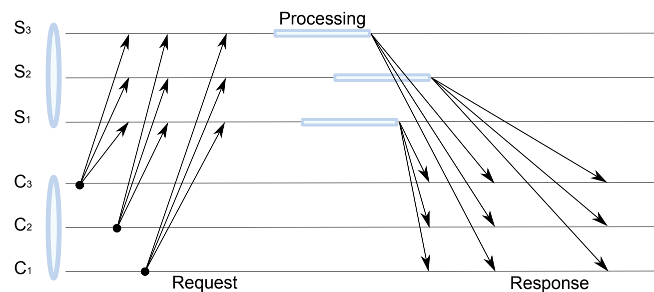

Figure 5. Groupcast binding

4.2. *Deterministic failure detection*

Our second challenge is failure detection because failure detectors are both imperfect and non-deterministic. As non-deterministic, a failure detector does not seem to qualify for an active replication scheme. Interestingly, our multigathercast semantics provides an elegant solution to the non-deterministic nature of failure detection, allowing us to replicate our Failure Detector like any other JADE subsystem. Our setting is the following. Each replica of the Failure Detector of a tile watches all nodes in that tile. As depicted in the Figure 6, each replica sends failure reports to all replicas of our Repair Manager through a multigathercast reference. Our multigathercast semantics forces a deterministic outcome since it absorbs redundant messages based on their sequence number and not their actual contents. Therefore, different failure reports that are multicasted with the same sequence number from different detectors will be absorbed. To the Repair Manager replicas, it appears as if all detectors had multicasted the same failure report, hence the deterministic outcome.

This design 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

Copyright © 2010 John Wiley & Sons, Ltd.

Prepared using speauth.cls
DOI: 10.1002/spe

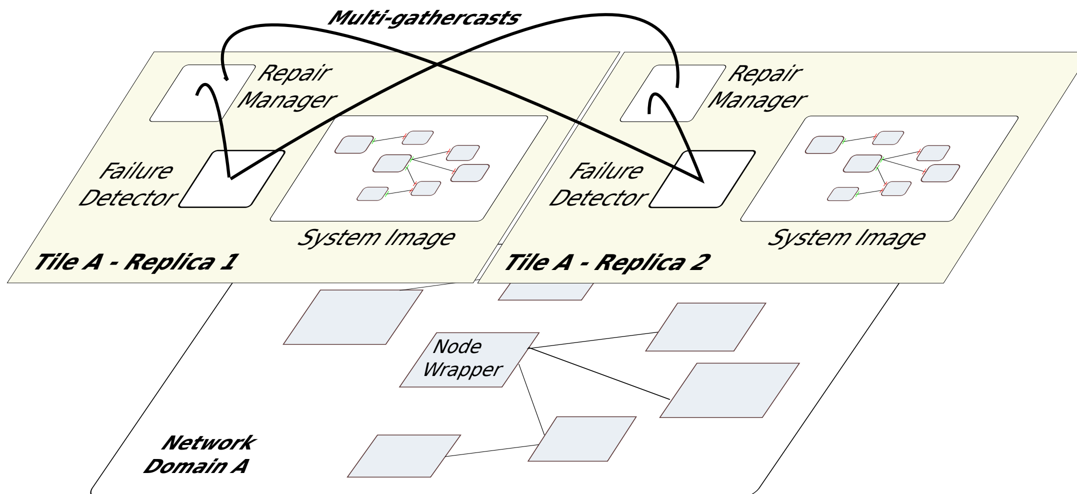

Figure 6. Replicated Node Failure Detector

Repair Manager. Conversely, this design does not prevent false positive (imperfect failure detector) but they are harmless. On the one hand, correct nodes detected as failed will be unbound from the rest of the distributed system by the next repair session. On the other hand, they can quickly commit suicide upon realizing that they have been detected as failed.

Due to space restrictions, we will only briefly discuss network partitioning. We assume no network partitioning within a tile as more and more network domains advocate hardware-redundant topologies. Between tiles, network partitioning results in aborting any repair session that tries to extend its impacted tile set across a network partitioning boundary. The repair session is only delayed as it will be retried later, once the network partitioning has been resolved.

4.3. *Repairing* JADE *Failures*

Active replication hides failures of JADE subsystems, however failures do happen and failed replicas must be detected, repaired, and reinserted in order to maintain the ability of JADE to resist its own failures over time. JADE ensures this self^2-repair ability through a recursive design whose cornerstone is the fact that JADE subsystems are wrapped like any other managed subsystem. Therefore, the wrappers of JADE subsystems composing a tile, called JADE *wrappers*, are mirrored in the System Image of their own tile, like regular managed wrappers, as depicted in Figure 7. The lower plane shows the Managed Distributed Environment that includes all managed wrappers, including the JADE wrappers. The top plane represents the System Image with one mirror per wrapper, including the mirrors of JADE wrappers. There is in fact one mirror and one wrapper per replica of a JADE subsystem.

This recursive design is transparent to the Failure Detector and the Repair Manager. Indeed, failure detection, analysis, and repair happens as they did before. Each replica of the Failure Detector watches over JADE subsystems like it watches over any managed subsystem. Each replica of the Repair Manager receives failure reports about failed JADE subsystems like any other failure report. It analyzes these failure as before, computing the list of impacted mirrors. It determines how to repair each impacted mirror, deciding to restart or relocate it. In other words, the entire repair session is unaware that it repairs JADE subsystems or not. The only impact of the self^2-repair property on the JADE design concerns the commit of a repair session when repairing a JADE subsystem replica. There are two specific design points to consider.

First, both sides of a reference between a JADE wrapper and its mirror are now replicated. Indeed, mirrors are replicated because they belong to the System Image that is replicated. Wrappers of JADE subsystems are replicated because the JADE subsystems are replicated. This seems to suggest the use of a multigathercast semantics between these two sides, when forwarding management operations

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

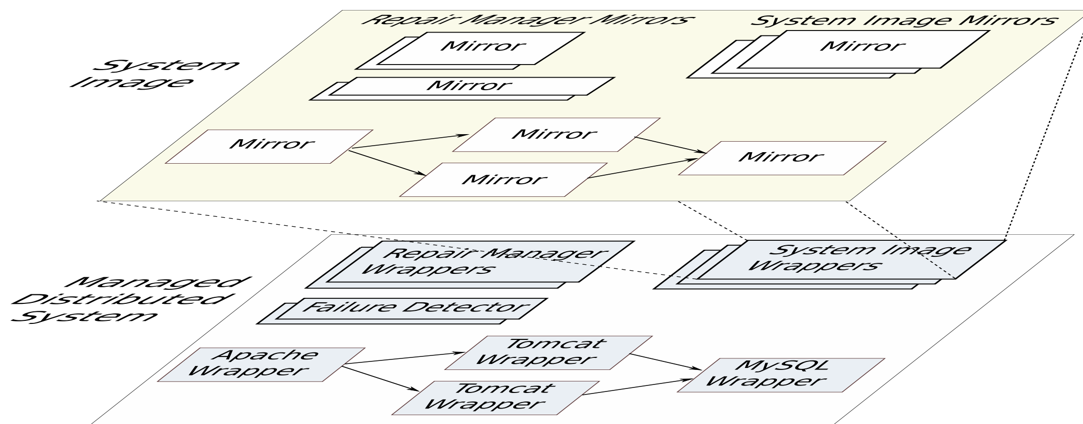

Figure 7. Recursive Design

from a mirror to its wrapper during the commit. In fact, this is quite the contrary, it is important to use a gathercast semantics rather than a multigathercast one. Indeed, when a JADE subsystem replica fails, repair operations are not intended for the entire replication group of the JADE wrapper but to the individual replica that has failed in that group. This is because we are repairing and reinserting individual wrappers in their replication group.

The second point concerns the reinsertion protocol of repaired wrappers replicas. Our solution is based on the traditional *View Synchronous Multicast* (VScast) [18]. The view change naturally happens at the *endApply* that represents the final step of a session commit. It is therefore important that repaired replicas reinsert themselves right after the commit that repaired them. This means that a repaired replica of the System Image reinserts itself with the state of the System Image that indicates that it has been repaired and reinserted. For the Repair Manager, its state being essentially a list of pending failures to analyze and repair, the reinsertion is straightforward.

5. EVALUATION

In this section, we evaluate our approach, discussing two illustrative contexts to which JADE brings autonomic repair—multi-tiered web servers and Network File Systems (NFS). These two contexts enable us to discuss both the wrapping of legacy systems and the performance characteristics of our solution. All experiments have been conducted with the following setting. Each legacy subsystem ran on a dedicated machine. We use two replicas of JADE. Machines were running the Linux kernel, version 2.6.22.14, interconnected using a 100 Mb/s Ethernet LAN. Each machine was a dual-core, Pentium T2300, at 1.66GHz, with a cache size of 2048KB, and 2GB of main memory. We used the Sun's Java Runtime Environment (JRE) 1.6 with the Apache Felix implementation of OSGi[‡] for our Java deployment, Java wrappers and JADE subsystems. As previously said, our multigathercast implementation relies on JGroups[§], a toolkit that provides group membership services and a reliable totally-ordered message delivery to dynamic groups. Our Node Failure Detector uses a simple Java-level ping every 10 seconds per watched node; advanced implementations are discussed in [7].

5.1. Web servers

In this experiment, we consider the autonomic repair of a Web application server. We used the RUBIS benchmark, using Apache 1.3.29 for the HTTP daemon, JONAS4.8 (Servlet engine and

[‡]http://www.osgi.org
[§]http://www.jgroups.org/javagroupsnew/docs/index.html

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

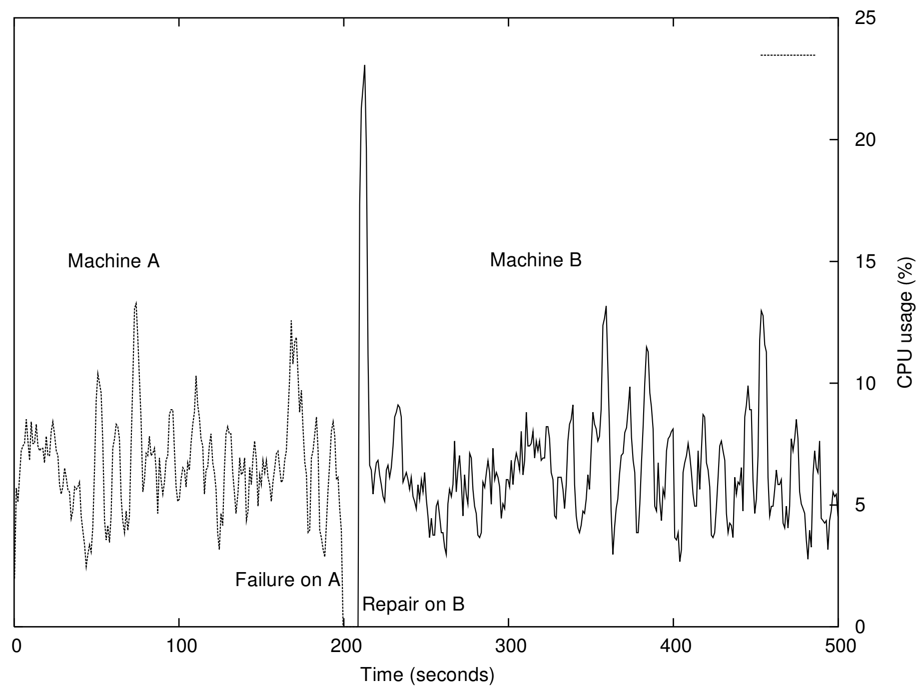

Figure 8. Repairing an Apache failure

EJB container)[¶] for the application server, and MySQL 4.0.17 for the database. The distributed setting was the following: each tier ran on a dedicated node (3 nodes plus one spare) and JADE was running on two other dedicated nodes plus one spare node.

First, we evaluated the performance overhead introduced by JADE. For this, we compared the performance of RUBIS, running standalone and running under the management of JADE. On a multi-tiered JEE system, the introduced overheads are negligible. The throughput and response times remain unchanged. We noticed a slightly larger memory usage (a couple of percents). Second, we evaluated the ability of JADE to repair failures. For this, we experimented with failures of either Apache or Tomcat in both a basic multi-tiered architecture and a clustered architecture designed for high availability.

Figure 8 depicts the failure and repair of an Apache daemon, across two machines. The two machines have synchronized clocks, allowing us for this experiment to speak of a global time line at the granularity of a second. The failure happens on one machine at the sharp complete drop in CPU usage, at time 200 seconds. We clearly see the delay of 10 seconds before the failure is detected and JADE starts repairing it. On the second machine, the sharp increase in CPU usage is the repair process that lasts about 6.2 seconds. This time includes the time of the repair session, including the time to commit. The commit involves 72 multigathercast invocations, accounting for 5 seconds. The remaining 1.2 seconds is mainly about starting Apache (0.5 seconds) and the deployment time of the Apache wrapper using OSGi, on an already running JRE.

Figure 9 depicts the failure and repair of a Tomcat server, in the context of a clustered enterprise server where each tier is replicated to ensure better availability and scalability. We used Apache with

[¶]http://jonas.objectweb.org

Copyright © 2010 John Wiley & Sons, Ltd.
Prepared using speauth.cls

DOI: 10.1002/spe

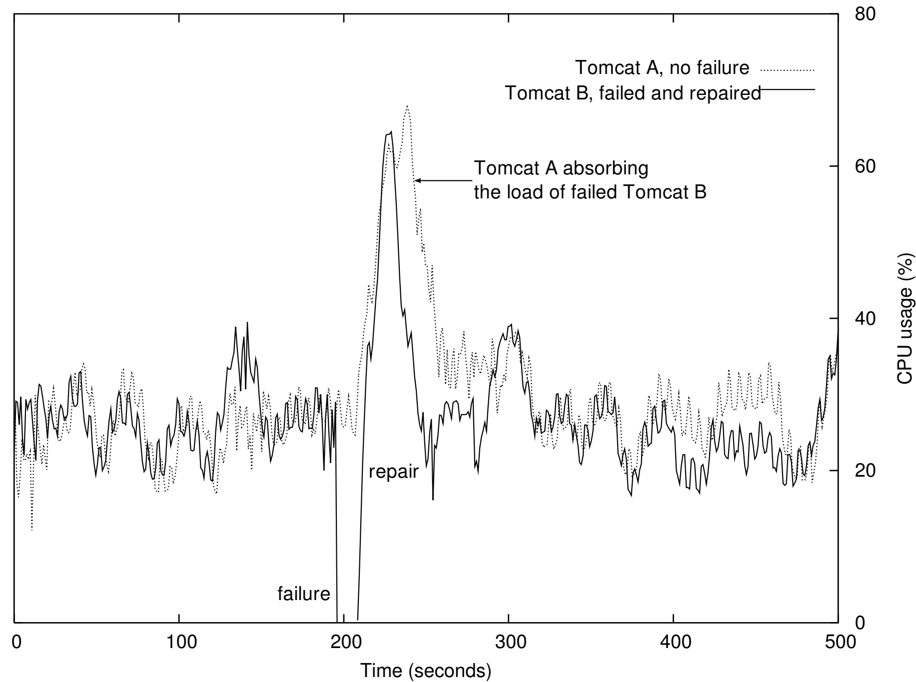

Figure 9. Repairing a Tomcat failure

the *modJK* plugin‖ (version 1.2), a clustered Tomcat (JONAS4.8) and C-JDBC [20], version 2.0.2, as the database server clustering system. The *modJK* plugin load balances requests on multiple Tomcats, providing high-availability with respect to Tomcat failures. The distributed setting was the following. The Apache HTTP daemon and MySQL ran on dedicated nodes. We used two Tomcats, running on dedicated nodes. JADE was still running on two dedicated nodes.

The pattern of the CPU usage shows the failure happening just before 200 seconds. Again, the delay of 10 seconds for JADE to detect the failure is clearly visible. Within these 10 seconds, the *modJK* plugin detects the same failure of Tomcat B and re-routes the entire load on Tomcat A, which produces the sharp increase in CPU usage from 30% to 60% on the machine hosting Tomcat A. The start of repair on the machine where JADE relocates Tomcat B is again clearly visible with the sharp increase of CPU usage. The time to repair is much longer, around 40 seconds once the failure has been detected. The overhead of deployment and multigathercast invocations, about 17 seconds, is much larger this time since we relocated both a JONAS and its RUBIS application. Even more expensive is the cold start of the relocated subsystems, JONAS and RUBIS, that accounts for the remaining 23 seconds.

While a downtime of about 50 seconds could be considered long, it has no real impact. Indeed, this downtime does not correspond to an interruption of service since we have a clustered Tomcat tier. There is however a short interruption of service when the Apache HTTP daemon is restarted by its wrapper. This restart is only necessary because of the *modJK* plugin that does not support hot reconfigurations. The only possibility to force *modJK* to reload its configuration and thereby considering the newly relocated Tomcat is to restart the Apache HTTP daemon, which takes less

‖http://tomcat.apache.org/connectors-doc/

Copyright © 2010 John Wiley & Sons, Ltd.

Softw. Pract. Exper. (2010)
DOI: 10.1002/spe

than 0.5 seconds.

The essential point is that JADE provides safe and automated repair without hindering the performance of the managed systems, even in the presence of clustered high-availability solutions. This is possible because JADE wrappers may actually delay and re-order management operations between the *beginApply* and *endApply*, as we explained in Section 2.1. In this very instance, the Apache daemon is kept running by its wrapper throughout the entire commit of the repair session, that is, while the failed Tomcat is repaired and restarted. The HTTP daemon is only stopped and restarted by its wrapper at the very last moment— the *endApply* invocation. The interruption of service is therefore minimal.

In the same context of a clustered JEE, we also experimented with the self^2-repair behavior of JADE. We forced the combined failure of one Tomcat and one complete replica of JADE. More precisely, we forced the failure of one replica of the Failure Detector, of the System Image, and of the Repair Manager. In this more complex situation of combined failures, the repair session, including the time to commit, takes about one hundred seconds. As previously mentioned, the repair of a failed Tomcat is about 50 seconds, which leaves about 50 seconds for the repair of the lost replicas of JADE subsystems. The time to deploy JADE wrappers and subsystems through OSGi is not significant here, the overhead is due to the number of multigathercast invocations when repairing the multiple JADE subsystems. Indeed, our implementation of JADE wrappers heavily relies on the use of attributes and each setting of an attribute on a subsystem's wrapper requires one multigathercast.

To confirm this, we carried specific measures of our multigathercast implementation on JGroups. Table I summarizes the results, times are in milliseconds for 10,000 invocations. In JADE, we use the Java proxy framework for our stubs and skeletons, so we first measured its local overhead. We then measured the overhead of using Java proxies combined with our caches based on sequence numbers but with a regular unicast semantics. We are about 60% slower than regular Java RMI. The third set of measures is the overhead of our complete multigathercast, including the overhead of JGroups. We are about 300 times slower than regular Java RMI, with about 70ms per multigathercast invocations. Each multigathercast is 4 multicasts, totally-ordered and reliable over a dynamic group of 4 members. Indeed, for each multigathercast, we have 2 clients sending method invocations and 2 servers sending back responses. Comparable times for JGroups, with UDP over 100Mbps Ethernet, can be found in these performance analysis [2][5].

		Method	Time(ms)
Java	Unicast RMI	void print()	1763
		void print(Object,Object)	1963
		Object print(Object, Object)	2313
JADE	Proxy	void print()	160
		void print(Object,Object)	141
		Object print(Object, Object)	150
	Unicast	void print()	2814
		void print(Object,Object)	3125
		Object print(Object, Object)	3185
	Groupcast	void print()	718683
		void print(Object,Object)	702830
		Object print(Object, Object)	704823

Table I. Groupcast Measures

Future work includes improving the performances of our solution, but although one hundred seconds may seem quite a long repair time for combined failures, this number needs to be put in perspective. Firstly, a failure of one or more JADE component does not prevent JADE from repairing the managed system or itself, even if they happen during a repair session. Secondly, we are still orders of magnitude faster than a manual repair by a skilled operator. Thirdly, our repair sessions do not introduce unavailability periods for highly-available clustered servers, as explained earlier, reinserting lost servers in less than a minute.

Copyright © 2010 John Wiley & Sons, Ltd.

Prepared using speauth.cls
DOI: 10.1002/spe

5.2. NFS

In this scenario, we consider the autonomic repair of an NFS infrastructure, as an example of a storage service in the Cloud. We experimented JADE with the autonomic repair of NFS servers replicated using *DRBD*. We wrapped NFS servers, NFS clients, and NFS applications, so that all subsystems are monitored and repaired. In particular, we support client machines that dynamically join or leave the managed distributed system, automatically mounting and unmounting exported NFS file systems. Applications handle the unavailability of the NFS filesystem by simply waiting, cognizant of the autonomic repair of servers. Applications keep updated data in memory and delay their save operation for the few seconds of unavailability.

Using JADE, the administrator starts with defining a pool of nodes for hosting the multiple NFS servers, called *NFS-server nodes*. The goal is to create several NFS servers with the same contents, organized as master and slaves. Each NFS server replica is associated with a *NFS-server mirror*. *NFS-server wrappers* contains code that wraps both the NFS server daemon and the exported disk partition. NFS-server mirrors are automatically created in the System Image, upon the creation of *NFS-server wrappers*. In our design, each NFS-server wrapper is bound to all other NFS-server wrappers. Furthermore, the names of the NFS servers must provide a complete ordering, effectively ranking NFS-server replicas.

This order is used by the NFS-server wrappers for the election of the master NFS server. At creation time, the wrappers will choose the server with the highest rank. When the current master fails, each wrapper sees a *unbind-bind* pair of operations for the binding to the failed master. It will also see such unbind-bind pairs for all bindings to other failed NFS-server wrappers, if any. With this knowledge, each wrapper independently elects the new master as the NFS-server wrapper that has the highest rank and that was not unbound in the current session. Since all wrappers apply the same algorithm, the new master will know it is the new master. This election happens during the *endApply* invocation. The new master will declare itself as the DRBD primary. It will start the NFS daemon, mount the disk partition, and export it.

In our experiment, client machines are laptops, dynamically joining or leaving the distributed system as they use or not the storage service. Each laptop runs the JADE platform, hosting a particular JADE subsystem that listens for network connectivity. When network connectivity is available, this subsystem attempts to connect to the System Image to remotely create an NFS-client mirror, binding it to all the appropriate NFS-server mirrors. These bindings are created using the NFS-server names, therefore having the very same order as the bindings between NFS-server mirrors. When committing this reconfiguration, JADE creates the NFS-client wrapper on the laptop, configures it, and binds it to the correct NFS-server wrappers. Since each client has the same ordered list of bindings to NFS-server wrappers and that it sees the same *unbind-bind* operations for any failed NFS server, including the master, it always knows about the current master, using the above election algorithm.

Given this knowledge, the rest of the wrapping is pretty straightforward. Each *unbind* operation from the current master corresponds to a local forceful unmount from that master (something that requires Linux kernel 2.1.116 or later). The new mount is done on the new master at the *endApply* invocation. This result in NFS clients failing over to a new master, which happens within 18 seconds in our experiments, 10 seconds for the failure to be detected and 8 seconds to actually repair. The NFS and DRBD operations account for less than a second. Our multigathercast invocations account for a little over 6 seconds. The remaining one second is the deployment time of the NFS-server wrapper using OSGi, on an already running JRE.

6. LESSONS LEARNED

Through the JADE experience, we gained a better understanding of the difficult challenge that reaching fully autonomic systems is. Through several prototypes and a five year period, we learned a few lessons that we share in this section, discussing the strengths and limitations of our solution.

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

6.1. Strengths

We believe that the JADE design exhibits several major strengths.

First, our work validated the use of a recursive approach to reach complete self-management, meaning not only a self-repair ability but also a self^2-repair ability. We therefore believe that JADE brings a significant contribution to the field of autonomic systems, with a particular focus on the challenges of a recursive approach. Self-management systems often provide useful features for managing distributed systems, but they are themselves complex distributed systems that need to be deployed and managed as well.

Second, JADE demonstrates the strength of an architecture-based approach for self-repairing distributed systems. Without this principled approach, we would not have been able to design an elegant solution for the self^2 repair problem.

Third, our solution validated the use of wrappers to build an homogeneous management infrastructure over heterogeneous legacy systems. The wrapping process is most often very straightforward, even across legacy systems as diverse as JEE applications [1], JMS systems [6], or NFS servers. With wrappers in place, JADE just works—encapsulating all the complexities of an autonomic repair. We therefore believe that we have successfully reduced to a minimum the amount of code that has to be tailored to managing specific legacy systems. The approach has been transferred out of our laboratory into an open source self-management infrastructure called JASMINe[**].

Fourth, JADE recovers failures more safely than skilled operators could. In 2003, a study [26] of the causes of failures of internet services perfectly illustrates this. The study found that operator error is the largest cause of failures, operator error is the largest contributor to time to repair, and configuration errors are the largest category of operator errors. The errors are exactly what the kind of errors that an autonomous repair system such as JADE eliminates.

Fifth, JADE recovers failures much faster than skilled operators ever could. Many failures are repaired in less than 1 minute, which includes the time to detect the failure, the time to deploy the software on a new node, the time to create wrappers, and the time to configure and start wrappers. While a MTTR of a minute could be considered high, it is still orders of magnitude better than what any human administrator can do. Additionally, JADE is compatible with the cluster-based design of high-availability JEE servers.

Sixth, JADE scales on large distributed environments, allowing managed systems to spread over wide area networks. The time to repair failures mostly depends on the number of failed subsystems and on how many tiles they are spread over. This time does not really depend on the actual size of the distributed environment hosting the tiles, beyond the obvious increase in latencies of the underlying network. Considering for instance a set of components C_1,...,C_n that are impacted by the failure of a component C_0, the time to repair will be composed as following.

- The time needed to detect the failure of C_0, something that is local to the tile of C_0.
- The time needed to lock (and later unlock) the tiles hosting a component in C_1,...,C_n, which involves two multigathercast communication by involved tile.
- The time needed to analyze the failure and reconfigure the mirrors, which includes $n*k$ multigathercast communications with the mirrors of C_1,...,C_n, where k corresponds to the average number of reconfiguration orders invoked on mirrors.
- The time to relocate subsystems, a time that may represent a significant part of the overall time to repair.
- The time needed to apply the reconfigurations onto the wrapped subsystem, a time that is mostly dependent on the legacy systems and how quickly it can be manipulated by its wrapper.

JADE also scales on large distributed environments because it is able to process non-conflicting repair sessions in parallel in different tiles. Typically, two JEE Application Servers running in distinct tiles could be repaired in parallel. However, JADE will serialize any two repair session that

[**] http://wiki.jasmine.ow2.org/xwiki/bin/view/Main/WebHome

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

need locking common tiles. Given our short repair times and the comparatively larger mean time to failure, we do not expect a high-level of concurrent failures. Also notice that a repair session does not suspend our fault detectors, so failures that would occur during a relatively long repair session would be detected, accumulated, and then processed altogether in a follow-up repair session. Additionally, many distributed systems deployed in Clouds are still small or medium size [16], spanning only a few tiles, thereby reducing the risk of tile-level contention during our time to repairing failures.

6.2. Constraints and limitations

Our solution has some limitations and imposes certain constraints on the kinds of distributed systems that can be managed and self-repaired.

One limitation is the scalability of the approach in terms of the size of managed distributed systems, that is, the number of managed subsystems. We started JADE for JEE cluster environments that have a rather small number of subsystems. We continued with Cloud environments, but also with a fairly small number of subsystems [16], but that span wide-area networks. More work is necessary to determine the scalability of our design when the number of subsystems increases drastically. We expect that the self-repair design will scale, that is, the wrapper and mirror approach can scale since we have one wrapper per subsystem and a in-memory Java representation of the architecture. We do not expect the time to process the reconfiguration of the System Image would be significant even with large systems.

However, we would expect that the commit time would increase significantly as the size of the managed systems would increase. The main overhead would come from using remote method invocations to issue reconfiguration operations on remote wrappers. The number of these operations grows as a linear function of the number of wrappers impacted by a fault, but also as a linear function of the number of bindings between impacted wrappers and the rest of the system. At one multigather cast per issued operation and with each multigathercast costing about 70ms, we expect the commit time to rapidly become quite significant. Future work should consider this a priority, with a possible optimization that could be strive for a parallel execution of the commit, issuing as much operations as possible concurrently on wrappers.

Aside from the above scalability limitation, we also impose that subsystems be designed as fail-stop subsystems. In other words, we do not handle Byzantine failures. However, wrappers can easily handle hung or very-slow subsystems. Despite the fact that different legacy systems would require different specific techniques, we can list a few straightforward techniques that could allow a wrapper to deduce that its subsystem is hung or very-slow. First, the wrapper could monitor the process performance in terms of CPU and memory usage. Second, the wrapper can fake a client, monitoring that its subsystem still answers to client requests. Upon suspecting its subsystem, a wrapper would simply kill it, forcing a failure that will be repaired as any naturally-occurring failure would.

The way JADE repairs node failures requires that the environment be capable of dynamically allocating nodes. Of course, these nodes can be real machines or virtual ones. This facility is provided by clusters or clouds, but not by Grids. Nevertheless, Grids environments can be used if a sufficient set of nodes can be pre-allocated to JADE and later used on demand internally. This is obviously not optimal if the Grid is shared. A related constraint is the ability to deploy the legacy subsystems on the allocated nodes, which is obviously greatly reduced when using virtual machines. Also, using virtual machines opens up the possibility to multiplex many nodes on fewer machines.

Finally, as mentioned previously in the paper, we assume that there are no network partitioning within a tile; but remember that we allow partitioning between tiles. A partitioned tile would only delay the repair sessions that would need to lock that tile. Within a tile, which is essentially a local-area network, we believe that our no-partition assumption is reasonable [24] since most local-area networks advocate hardware-redundant network topologies.

To conclude, we discuss shortly some specific limitations of JADE that are not inherent to the design of JADE and therefore represent interesting perspectives for future work:

- The repair policy of JADE is to repair a system by restoring it as it was previously to the failure. In other words, JADE restores the managed systems to a software architecture that is equivalent to the architecture it had prior to the failure. We haven't considered alternative

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

policies such as different notions of architecture equivalence, may be considering different quality of service. This could help in the presence of recurring failures, a situation that defeats our current policy. At the very least, we would need to detect that we entered a cyclic failure-repair pattern and ask for a human intervention.
- When there are no more free nodes available in the execution environment to relocate failed subsystems, JADE can no longer restore the managed systems to a software architecture that is equivalent to the architecture it had prior to the failure. A short-term way out is just to accept that some failed subsystems are lost because they cannot be relocated, which in turn means that some other subsystems might no longer have all their dependencies satisfied. Later on, as some nodes may become available, it would be possible to attempt the relocations then, in some sense finishing the repair that couldn't finish previously.
- Human administrators need tools to observe JADE: the failures it detects, the decisions it takes, and the reconfigurations it carries on. Tracing and logging techniques are certainly a possible step in the right direction. As always, the challenge would be to know what to log, try to cap the amount of information logged. Also, logging low-level reconfiguration operations might be overwhelming for most administrators, raising the question of higher-level reconfiguration goals that would be meaningful for human administrators. Certainly, this concept of goals would need to be related to the possibility to identify different repair policies.

7. RELATED WORK

Fault tolerance and repair have been the subject of much work, from simple to complete software solutions. At one end of the spectrum, simple solutions exist to watch and repair individual processes, such as the inet daemon or more generally watchdog mechanisms. While simple and efficient, these solutions only address the repair of individual systems rather than the repair of a distributed system composed of multiple cooperating subsystems.

More specific to distributed systems, dedicated repair solutions have been proposed, taking into account subsystems dependencies. Chameleon [4] proposes that distributed applications be written to the Chameleon runtime or that special hooks be inserted in the operating system. Pinpoint [13] instruments the JEE middleware to track requests as they flow through the system, focusing on mining the collected paths to locate faults. The JBoss Application Generic Recovery (JAGR) [12][11] combines a failure path inference and micro-reboots. Micro-reboots are fast reboots that JAGR performs at the sub-application level to recover components from transient failures. These techniques are designed to be autonomous and application-generic, but they are specific to JEE servers. Our approach could similarly benefit from micro-reboots when the necessary application-level knowledge is available.

In contrast with dedicated solutions, generic and transparent repair facilities have been proposed based on virtualization techniques that provide failover based on the replication of a whole system [14]. Cruz [19] completes the approach of Zap [27] and deliver process-level migration without loosing TCP/IP connections. Remus [28] provides failover based on the replication of a whole system. Such solutions are entirely compatible with existing software, both applications and operating systems alike. These solutions are complementary to our self^2-repair proposal. They bring very interesting low-level properties for repair management that our solution can leverage. Conversely, these solutions can benefit from our solution to capture the architecture of the managed system and to provide the necessary control of the failover decisions and policies. In particular, our self^2 approach could be considered to ensure the autonomic repair of the failure detection and the failover management.

Closer to our approach that captures the distributed architecture of the managed system, several systems embodying an architecture-based approach to system management have been been proposed in the past few years, notably Automate [29], OpenORB [8], Rainbow [17] and Recover [3]. Such solutions provide an autonomic repair of the managed applications but provide no details on how the repair system repairs itself. In the case of Rainbow and Recover, the recovery decision process is centralized, making the management subsystem a single point of failure. The

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

papers on Automate and OpenCOM/OpenORB hint at a possible decentralization of the decision process but neither explain how this decentralization would take place nor how it would be made failure-tolerant.

8. CONCLUSION

We presented in this paper the full design of the JADE Autonomic Repair System that is uniquely suited for introducing truly autonomic repair for small to medium sized distributed systems, deployed in distributed environments such as clusters, clouds or grids.

A key design principle of our proposition is the use of the System Image that gives a representation of the software architecture of the managed distributed systems, including not only the description of the individual subsystems but also the description of their dependencies. Leveraging this representation, we achieve not only the usual self-repair ability but also the more unique self^2-repair ability. To achieve this, we first replicate the JADE subsystems for fault-tolerance. Second, we make JADE appear in the System Image as any distributed system it observes and repairs, allowing JADE to repair JADE using the very same approach it uses to repair any managed system.

Moreover, JADE does not require any modification of existing legacy systems, advocating a simple wrapping approach that leverages the same management capabilities a human administrator would use. Additionally, JADE scales through a simple yet efficient design based on tiling that naturally partitions the distributed environment into separate repair areas. Finally, JADE detects and repairs at the granularity of individual subsystems, taking human errors out of an increasingly complex repair process and delivering a far smaller mean time to repair that any skilled operator could.

REFERENCES

1.
2. T. Abdellatif, E. Cecchet, and R. Lachaize. Evaluation of a Group Communication Middleware for Clustered J2EE Application Servers. In *Int. Symposium on Distributed Objects and Applications*, 2004.
3. N. Arshad, D. Heimbigner, and A.L. Wolf. Deployment and dynamic reconfiguration planning for distributed software systems. *Software Quality Journal*, 15(3):265–281, 2007.
4. Saurabh Bagchi, Erik Haakenson, Keith Whisnant, Jun Wang, Xiao Yuxiao, and Mahesh Kalyanakrishnan. Chameleon: A software infrastructure for adaptive fault tolerance in distributed systems. 1998.
5. R. Baldoni, S. Cimmino, C. Marchetti, and A. Termini. Performance analysis of Java group toolkits: A case study. In *FIDJI '01: Revised Papers from the International Workshop on Scientific Engineering for Distributed Java Applications*. Springer-Verlag, 2003.
6. R. Balter. JORAM: The Open Source Enterprise Service Bus, 2004. `http://www.scalagent.com/pages/en/datasheet/040322-joram-whitepaper-en.pdf`.
7. M. Bertier, O. Marin, and Pierre Sens. Implementation and performance evaluation of an adaptable failure detector. In *International Conference on Dependable Systems and Networks*, 2002.
8. Gordon S. Blair, Hector Duran-Limon, Geoff Coulson, Paul Grace, Nikos Parlavantzas, Lynne Blair, and Rui Moreira. Reflection, self-awareness and self-healing. In *PROCEEDINGS OF THE FIRST WORKSHOP ON SELF-HEALING SYSTEMS*, pages 9–14, 2002.
9. S. Bouchenak, F. Boyer, D. Hagimont, S. Krakowiak, A. Mos, N. de Palma, V. Quéma, and J.B. Stefani. Architecture-Based Autonomous Repair Management: An Application to J2EE Clusters. In *24th IEEE Symposium on Reliable Distributed Systems*, 2005.
10. B. Burke and S. Labourey. Clustering With JBoss 3.0, oct 2002. http://www.onjava.com/pub/a/onjava/2002/07/10/jboss.html.
11. G. Candea, E. Kiciman, S. Kawamoto, and A. Fox. Autonomous recovery in componentized Internet applications. *Cluster Computing*, 9(2), 2006.
12. George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, and Armando Fox. A microrebootable system – design, implementation, and evaluation. *CoRR*, cs.OS/0406005, 2004.
13. Mike Y. Chen, Anthony Accardi, Emre K?c?man, Jim Lloyd, Dave Patterson, Armando Fox, and Eric Brewer. Path-based failure and evolution management. In *Proceedings of the international symposium on Networked Systems Design and Implementation(NSDI04*, pages 309–322, 2004.
14. C. Clark, K. Fraser, H. Steven, J.G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proc. of the USENIX Symposium on Networked Systems Design and Implementation*, 2005.
15. ANR (Agence Nationale de la Recherche). The Selfware Project, 2006-2009. `http://sardes.inrialpes.fr/selfware/`.
16. Xavier Etchevers, Thierry Coupaye, Fabienne Boyer, and Noel De Palma. Self-configuration of distributed applications in the cloud. In Ling Liu and Manish Parashar, editors, *IEEE CLOUD*, pages 668–675. IEEE, 2011.

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

17. D. Garlan, S.W. Cheng, A.C. Huang, B. Schmerl, and P. Steenkiste. Rainbow: Architecture-Based Self Adaptation with Reusable Infrastructure. *IEEE Computer*, 37(10), 2004.
18. R. Guerraoui and A. Schiper. Fault-tolerance by replication in distributed systems. In *International Conference on Reliable Software Technologies*. Springer Verlag, 1996.
19. G. Janakiraman, J.R. Santos, D. Subhraveti, and Y. Turner. Cruz: Application-Transparent Distributed Checkpoint-Restart on Standard Operating Systems. In *International Conference on Dependable Systems and Networks*, 2005.
20. Emmanuel Cecchet Julie, Julie Marguerite, and Willy Zwaenepoel. C-jdbc: Flexible database clustering middleware. In *In Proceedings of the USENIX 2004 Annual Technical Conference*, pages 9–18, 2004.
21. M. Kalyanakrishnam, Z. Kalbarczyk, and R. Iyer. Failure data analysis of a lan of windows nt based computers. In *Proceedings of the 18th IEEE Symposium on Reliable Distributed Systems*, SRDS '99, pages 178–, Washington, DC, USA, 1999. IEEE Computer Society.
22. J.O. Kephart. Research challenges of autonomic computing. In *Proc. of the International Conference on Software Engineering (ICSE)*, pages 15–22, New York, NY, USA, 2005. ACM.
23. J.O. Kephart and D.M. Chess. The Vision of Autonomic Computing. *IEEE Computer Magazine*, 36(1), 2003.
24. Joshua B. Leners, Hao Wu, Wei-Lun Hung, Marcos Kawazoe Aguilera, and Michael Walfish. Detecting failures in distributed systems with the falcon spy network. In Ted Wobber and Peter Druschel, editors, *SOSP*, pages 279–294. ACM, 2011.
25. Michael Murphy, Linton Abraham, Michael Fenn, and Sebastien Goasguen. Autonomic clouds on the grid. *Journal of Grid Computing*, 8:1–18, 2010. 10.1007/s10723-009-9142-3.
26. David Oppenheimer, Archana Ganapathi, and David A. Patterson. Why do internet services fail, and what can be done about it? In *Proceedings of the 4th conference on USENIX Symposium on Internet Technologies and Systems - Volume 4*, USITS'03, pages 1–1, Berkeley, CA, USA, 2003. USENIX Association.
27. Steven Osman, Dinesh Subhraveti, Gong Su, and Jason Nieh. The design and implementation of zap: a system for migrating computing environments. *SIGOPS Oper. Syst. Rev.*, 36:361–376, December 2002.
28. Steven Osman, Dinesh Subhraveti, Gong Su, and Jason Nieh. The design and implementation of zap: a system for migrating computing environments. *SIGOPS Oper. Syst. Rev.*, 36:361–376, December 2002.
29. M. Parashar, H. Liu, Z. Li, V. Matossian, C. Schmidt, G. Zhang, and S. Hariri. Automate: Enabling autonomic applications on the grid. *Cluster Computing*, 9(2), 2006.
30. S. Sicard, F. Boyer, and N. De Palma. Using components for architecture-based management: the self-repair case. In *Proc. of International Conference on Software Engineering*, 2008.
31. Zhuopeng Zhang and Hongji Yang. Incubating services in legacy systems for architectural migration. *Asia-Pacific Software Engineering Conference*, 0:196–203, 2004.

Copyright © 2010 John Wiley & Sons, Ltd.

Prepared using speauth.cls
DOI: 10.1002/spe