Using Components for Architecture-Based Management

The Self-Repair case

Sylvain Sicard
Université Grenoble I
INRIA Rhône-Alpes
sylvain.sicard@inrialpes.fr

Fabienne Boyer
Université Grenoble I
fabienne.boyer@inrialpes.fr

Noël De Palma
Institut National
Polytechnique de Grenoble
noel.depalma@inrialpes.fr

ABSTRACT

Components are widely used for managing distributed applications because they not only capture the software architecture of managed applications as an assembly of components but also permit to dynamically adapt these applications to changing environments. Following this approach, our practical experience in the JADE environment about developing an autonomic repair management service with a self-healing behavior shows novel requirements on reflective component models for architecture-based management systems. First, we have identified five essential runtime abstractions that a component model must include in order to efficiently support an autonomic repair service. Second, our experience suggests that traditional reflective component models should be extended so that it is possible to specialize meta-operations. Third, our experience also shows that a meta-data checkpointing capability is best-suited for meta-data recovery after failures. We demonstrate the soundness of these findings in several ways. We applied the difficult problem of autonomic repair to both J2EE and JMS middleware. We further stressed our algorithms and mechanisms by applying them recursively towards gaining a self-healing property for the repair service itself. Although our experience was done in the JADE context, using the FRACTAL component model, we believe our findings to be general to architecture-based management systems using reflective component models.

1. INTRODUCTION

Architecture-based management approaches promote the use of architectural models as guidelines for various management functions [13], [21]. Such approaches facilitate the building of autonomic management services which perform administration tasks without (or with minimal) human intervention. This trend has emerged as an important research agenda in the face of the ever-increasing complexity of distributed applications. Management services include application deployment, platform configuration, and reaction to events such as node failures, wide variations in load, various kinds of attacks, etc. An important part of this agenda lies in the elicitation of architectural principles and design patterns for the construction of autonomic systems.

A well-accepted design principle in system management consists in using a component-based technology to develop the management framework, such as in JMX [18], SMARTFROG [15, 22], JADE [7], RAINBOW [13], EGA [2]. Based on our experience in building a self-repair service in the context of an architecture-based management framework called JADE [7], we observed that the relevance of the repair service strongly depends on the properties and services provided by the component model. The contribution of this paper is to report on this experience and to expose these properties and services which are seldom satisfied by the existing approaches.

We more specifically consider the case of self-repairing J2EE clusters. Apart from the fact that J2EE application servers constitute an important and representative segment of distributed computing, there are two main reasons for this choice of application area and experimental setting.

First, a clustered J2EE application server constitutes a distributed system which has a non-trivial architectural complexity, involving several interacting tiers, and combining different legacy middleware technologies (Web servers, EJB servers, databases). Dealing with such levels of complexity has implications on the component model, which should provide the relevant abstractions to reify the software architecture of the managed legacy system.

Second, repair management in J2EE application servers still remains an open issue, witness several recent papers that deal with this subject [17, 23]. A self-repair service provides an entirely automatic repair process dealing with failures occurring either in the application or in the management system itself. To build such a service, additional properties have to be supported by the component model. More precisely, we have identified a need for associating the components with a meta-data checkpointing facility. Furthermore, some components had to be enhanced with replication capabilities.

The rest of the paper is organized as follows. Section 2 exposes the objectives of architecture-based management systems. Section 3 presents the basic properties expected from a component model used as a building block of an architecture-based management system. Section 4 focuses on the building of a repair management service and presents additional requirements that apply on the component model. The ideas exposed in this paper have been validated by an experiment whose results are exposed in Section 5. In particular, we give the principles of the FRACTAL [8] component model we used, as well as the way we enhanced it to fulfill the identified requirements. Related work is presented in Section 6, and we conclude in Section 7.

2. OBJECTIVES OF AN ARCHITECTURE-BASED MANAGEMENT SYSTEM

In this section, we present the objectives of architecture-based management systems, and the way components can be used in such systems.

2.1 Architecture-Based Management

When managing a distributed application, architecture-based management suggests to consider two levels: the application itself and the environment in which the application is deployed. Both are looked as composed of elements. The application is composed of software elements connected together through distinct kinds of relationships, such as composition or delegation. The environment is composed of hardware elements, such as nodes. Hardware elements are also connected through different relationships such as network connections.

Elements, both hardware and software, exhibit a *management state*. The management state of an element is composed of certain aspects exposed to the control of a management system by that element. Three such aspects are usually considered as essential in architecture-based management: local configuration settings (i.e., its properties values), life cycle state and relationships between managed elements. It is important to point out that the concept of management state is a dynamic one since aspects may change during the lifetime of the application.

As an example, we can consider the management state of an Apache Httpd server as exposing its current configuration settings (corresponding to properties defined in a file named `httpd.conf`), its life cycle state (started/stopped) and the host address and port of the Tomcat servlet server to which it is connected as a client (if such a connection exists).

Using the management states of elements, we can define the management state of a distributed application as whole that captures the current software architecture and configuration settings of a distributed application. The global management state is composed of the management states of the elements belonging to both the application and the environment. This includes the relationships between elements of the application and the environment, that is, how the application's elements are mapped on the environment's ones.

According to these definitions, two kinds of basic management functions need to be provided by an architecture-based management system. One must be able to observe the management state of the application, as an open box inside which the internal software architecture is made explicit. One must also be able to manipulate this management state, for instance creating or deleting elements as well as modifying their attributes or their relationships.

2.2 The J2EE Use Case

The J2EE specification aims at enabling the design of complex web application servers, which dynamically produce web pages in response to client requests. A J2EE application server is generally composed of four tiers, which can be deployed on different nodes:

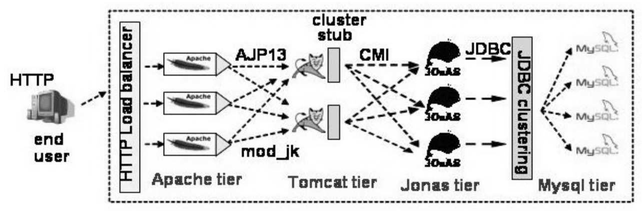


Figure 1: A Classical J2EE Application Server Architecture

- The web (e.g., Apache) and Servlet (e.g., Tomcat) server tiers execute the presentation part of the application, managing references to both static and dynamic web pages. Dynamic pages are generated using data that can be requested from the EJB tier.

- The EJB tier (e.g., JOnAS [20]) hosts the business logic of the application. If needed, the EJB server could interact with one ore more database servers (the fourth tier) which deal with persistent data used by the application.

Most J2EE application servers must be scalable (i.e., be able to deal with an arbitrary number of client requests in a reasonable time) and must ensure availability (i.e., be able to serve requests despite failures). Both scalability and availability can be achieved by replication of the different tiers on a cluster of machines, as illustrated in Fig. 1.

In this context, a typical management scenario consists in dealing with the failures of machines by (1) detecting such failures and (2) automatically restarting the failed replica on another node and updating its connections with the different tiers. Such a scenario implies that the management system should be aware of the software architecture of the application server and able to manipulate this architecture.

2.3 Using Components

In the context of an architecture-based management system, the components may be used in two ways:

- As wrappers of legacy elements: components are used to reify the management interface of the legacy elements (and not their functional interface) into a *uniform management interface* providing operations for observing and manipulating their management state. Most management services can then be designed in a generic way, independently of the specific management interface of the managed legacy elements.

- For building management services: the expected benefit is that the management services can be applied to the management system itself. For instance, a self-sizing service can be under the control of a repair management service in order to be automatically repaired in case of its failure.

The requirements related to the building of wrappers are presented in Section 3 and those related to the building of a repair management service are presented in Section 4.

3. USING COMPONENTS AS WRAPPERS OF LEGACY ELEMENTS

This section presents the main requirements that apply to a component model used for wrapping legacy elements in an architecture-based management system. We suppose that the basics of component models [24] are well-known.

3.1 Objectives

A relevant design principle for an architecture-based management system is to represent the management state of a legacy application using the abstractions (e.g., interfaces, bindings, etc.) provided by the component model. These abstractions are indeed close to those required to represent a management state. Following this principle, a wrapper performs the two following tasks.

- It reifies the management state of a legacy element into the component model abstractions. For instance, a legacy element is reified into a component, and a composition link is reified into a containment relation between components.
- It provides a uniform way of manipulating this management state through the meta-operations (e.g., bind, unbind, etc.) associated to the component model. In other words, the management operations correspond to the meta-operations. In the case of a composition link between two legacy elements, modifying this link should be performed by invoking the meta-operations manipulating the containment relation which reifies this link, given that the corresponding legacy actions are executed at the legacy level.

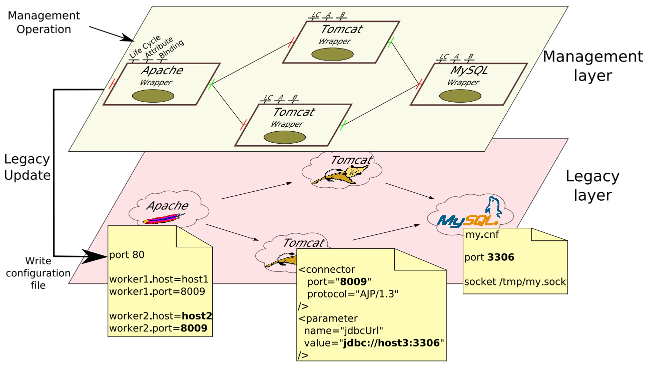
This leads to an overall architecture composed of two layers, as shown in Fig. 2. Since management state is a dynamic notion, only component models providing runtime components can be considered here. Consistency maintenance between the two layers should ensure that the Management layer reflects the current management state of the Legacy layer. In principle, any manipulation of the Legacy layer must go through the control of the management system.

Reaching these objectives implies that the abstractions provided by the component model are relevant to represent the management state of a legacy element, and that the meta-operations used to manipulate these abstractions are sufficient. The next sections describe more precisely the expected abstractions (Section 3.2) and meta-operations (Section 3.3).

3.2 Requirements for the Component Model Abstractions

Our experience has shown that the five following abstractions are sufficient to reify the management state of a legacy element (also called *managed element*), with respect to the requirements of a repair management service.

- **Attributes**. The attributes of a component reify the configurable properties of its managed element (e.g., properties defined in the Web server's configuration file).
- **Life cycle state**. The life cycle state of a component reifies the life cycle state of its managed element (e.g., deployed, started, etc.). This implies that the set of possible states defined by the component model be extensible to reflect the states of the Legacy layer.



```
port 80

worker1.host=host1
worker1.port=8009

worker2.host=host2
worker2.port=8009
```

```
<connector
  port="8009"
  protocol="AJP/1.3"
/>
<parameter
  name="jdbcUrl"
  value="jdbc://host3:3306"
/>
```

```
my.cnf

port 3306

socket /tmp/my.sock
```

Figure 2: Basic Architecture of a Management System

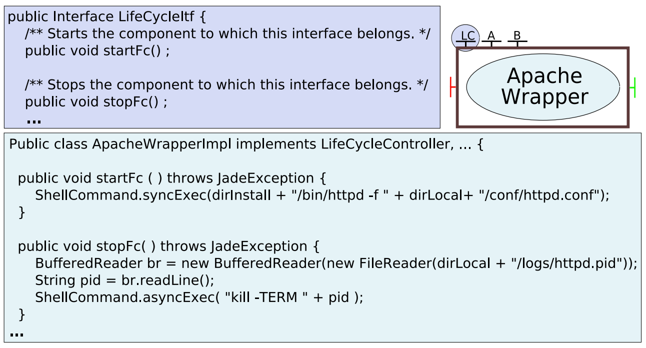
- **Interfaces**. The interfaces of a component reify the functional dependencies of its managed element with the other managed elements in order to provide control on an application's configuration. For instance, two legacy elements such as an Apache Httpd Web server and a Tomcat Servlet server may be wrapped by two components having respectively a client and server interface of a compatible type, meaning that the two servers can be connected together by the management system.
- **Bindings**. Bindings between components reify connections between managed elements that have functional dependencies. An important point is that, by reification, bindings provide a way to manage connections (e.g., creating or closing connections), but they are not intended to implement them.
- **Containment**. Containment relations reify architectural composition links between legacy elements. For instance, a J2EE component is a composite one, to reflects the software architure illustrated in 2.2.

It should be noted that the mapping between the software legacy elements and the hardware ones (e.g., nodes) can be represented either by binding or by containment. In the later case, the hardware elements of the execution environment are wrapped into composite components which contain their consumers as sub-components (e.g., a node component contains sub-components representing the legacy elements running on it). Using the containment feature to represent the mapping between the execution environment and the legacy application requires, however, that the component model should allow components to be shared among multiple parent components.

3.3 Requirements for the Component Model Meta-Operations

Given our five abstractions and the way they are used to represent a legacy system, the two following classical kinds of meta-operations are required:

- **Introspection**. Introspectable components provide a way to dynamically discover their meta-data. By



```
public Interface LifeCycleItf {
   /** Starts the component to which this interface belongs. */
   public void startFc() ;

   /** Stops the component to which this interface belongs. */
   public void stopFc() ;
   ...
```

```
Public class ApacheWrapperImpl implements LifeCycleController, ... {

  public void startFc ( ) throws JadeException {
     ShellCommand.syncExec(dirInstall + "/bin/httpd -f " + dirLocal+ "/conf/httpd.conf");
  }

  public void stopFc( ) throws JadeException {
     BufferedReader br = new BufferedReader(new FileReader(dirLocal + "/logs/httpd.pid"));
     String pid = br.readLine();
     ShellCommand.asyncExec( "kill -TERM " + pid );
  }
...
```

Figure 3: Wrapping the life cycle interface of the Apache legacy software.

meta-data, we means the current state of our five abstractions. This feature allows observing the management state of the legacy application.

- **Reconfiguration**. Reconfigurable components provide a way to dynamically modify their meta-data (e.g., adding / removing a binding). This indirectly allows modifying the management state of the legacy application, provided that the corresponding actions are reported at the Legacy layer.

While operations of this kind are often provided by reflective component models ([16]), a less frequently covered aspect is that the implementations of these operations should be **specializable**. This is required to manage the mapping with the Legacy layer through the specific management interfaces provided by the legacy elements. For instance, a web server and a database server can be wrapped into two components that provide a `setProperty` operation. While the former translates this operation into editing the web server's configuration file (e.g., for setting the server port number), the later may translate the same management operation into defining a parameter value used when starting the database server (e.g., for setting the maximum number of parallel connections). In the same way, when a `bind` operation is invoked on a wrapper component, the corresponding specific actions which establish the binding at the Legacy layer should be executed (e.g., creating a TCP connection). To avoid synchronization problems, a locking mechanism should of course be provided by the component model.

Figure 3 illustrates the wrapping of the Apache legacy software in the JADE system, and shows more precisely the specialization of the life-cycle operations to manage the mapping with the Apache legacy software.

4. USING COMPONENTS FOR MANAGEMENT SERVICES: THE CASE OF A REPAIR SERVICE

An autonomic repair service is one of the most complex management service and as such a very interesting case study of the architecture-based management approach and the abstractions required in the underlying component model.

The goal of an autonomic repair management service is to detect the occurrence of some well-identified types of failures and to restore a managed application to an active state. The goal is to achieve a specified level of availability, according to a given policy [4]. One policy consist in restoring the application to a known management state which existed prior to the failure.

The most challenging aspect of an autonomic repair is that it must not only repair the applications that it manages but also demonstrate a self-healing behavior. An interesting approach is to provide self-healing behavior through the same algorithms and mechanisms that were used for repairing managed applications.

In the following subsections, we present the additional requirements needed to build (1) a basic repair service and (2) enhance it with self-healing behavior.

4.1 Requirements for a Basic Repair Service

We consider the case of a repair service that restores an application's management state to the state which existed prior to the failure.

The core process executed after a failure detection involves the following main steps.

- **Analysis step**. Identify the failed elements and get their management state.

- **Substitution step**. Substitute the failed elements identified by the previous step by newly instantiated ones configured with the same management state.

Both steps involve the manipulation of the components of the Management layer (wrapper components), as explained in the two following sub-sections.

4.1.1 Meta-data checkpointing for the Analysis step.

The analysis step must introspect the wrapper components in order to get the knowledge allowing the failure to be repaired. The components concerned by the failure are those wrapping the failed node and the software elements which were running on it prior to failure. The required knowledge corresponds to the meta-data of these components (what are their bindings, containments, attributes, etc.). To be able to perform this introspection task, these components should be available, even after the failure.

One solution to ensure such property is to forbid a wrapper component to be co-located with its managed element. However, this is not realistic in the context of multi-tier distributed applications, where the execution time of the management functions can be critical (as for self-sizing services). Moreover, some managed elements (such as nodes) may not be entirely manipulated from a distant location, implying these to be co-located with their wrapper.

Thus there is a requirement for using a checkpointing mechanism which provides an up to date view of the meta-data of the wrapper components. This leads to the global architecture of the management system as illustrated in Fig. 4.

For uniformity reasons, the Checkpoint layer may be composed of components. The same introspection interface can then be used to introspect the Management and the Checkpoint layers. Following this principle, any *checkpointable* component is associated to a checkpoint component having the same type but an empty implementation part. The invocation of any meta-operation on such a component (e.g., `addSubComponent(...))` is then reflected at the Checkpoint layer (e.g., by establishing the corresponding containment relation).

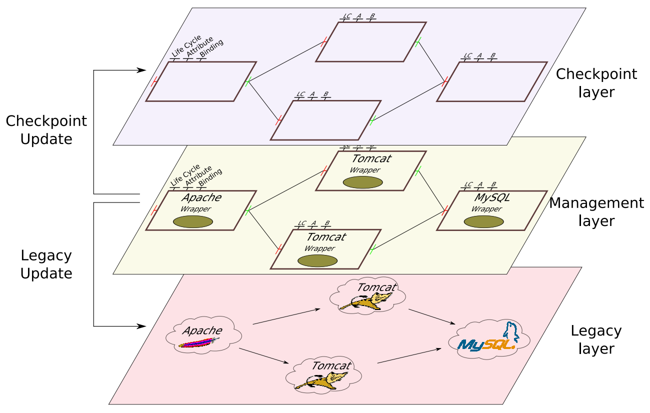


Figure 4: The Global Management Architecture

From a global point of view, invoking a meta-operation on a wrapper component (such as invoking a `bind(...)` on an Apache wrapper) causes the following actions:

- Checkpoint update: performing the same invocation at the Checkpoint layer, (e.g., establishing the corresponding binding at the Checkpoint layer),
- Legacy update: invoking the specific management interface of the Apache legacy component (e.g., establishing the binding at the Legacy layer), as said in 3.3.

4.1.2 *Component Reconfiguration for the Substitution step.*

The principle of the substitution step is to replace the failed components by new ones at the Management layer. The reconfiguration actions mostly concern the component bindings and containments. If a failed component was bound to other components, the bindings should be closed and re-established with the replacing component to reflect the same connections. Similarly, if a failed component was a subcomponent of other components, the containment relationships should be updated accordingly.

4.2 Requirements for adding Self-Healing Behavior

To achieve self-healing behavior for the above repair service, we need to add reliability to both the Management and Checkpoint layers:

- Add reliability the core process involved by the repair service as defined in Section 4.1
- Add reliability the critical data accessed by the repair service (the components of the Checkpoint layer)

A classical way for adding reliability is to use redundancy. By replicating both the core process of the repair service and the Checkpoint layer, a fault-tolerant repair service can be provided. However, the number of faults that are tolerated is limited by the replica cardinality. This is still insufficient with regard to a self-healing property, because each time a failure appears in the repair service, a human administrator must detect it and re-establish the replica cardinality.

In our approach, we avoid this human intervention. Because the repair service is able to repair itself, the repair

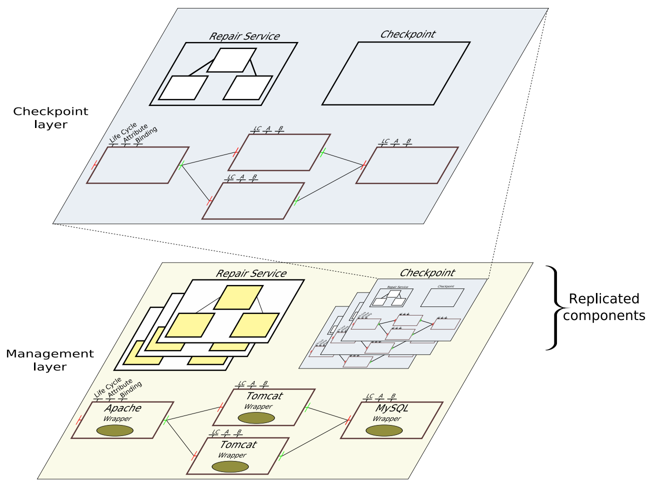


Figure 5: Putting pieces together: repair service & replicated components.

service automatically manages the re-establishment of the replica cardinality. Our approach achieves this without modifying (i.e., patching) the repair service. Indeed, since the repair service is implemented as components, the self-healing behavior of the repair service is obtained through the same algorithms and mechanisms as those used for repairing the legacy elements. This has the following implications.

- The replicas of both the core process of the repair service and the Checkpoint layer must be under the control of the repair service. Therefore, each replica should be represented as a component in the Management layer, which delimits the “repair” area (see 4.1.1).
- Replication should be provided as an orthogonal aspect allowing a component to be tagged with a “replicated” capability without having to program it specifically.

Since the Checkpoint layer provides an isomorphic view of the Management layer, the Checkpoint replicas will be visible in the Checkpoint layer itself. To prevent an infinite recursion in the Checkpoint layer, the meta-operations of the Checkpoint components are specialized in a way that avoids the *Checkpoint update* action presented in 4.1.1.

The resulting overall management architecture is illustrated in Fig. 5. The operating principle to gain self-healing capability is as follows: (1) the components implementing the repair service (including nodes failure detector) as well as Checkpoint components are *replicated* components; (2) these replicas are represented (i.e., referenced) in the Management layer; (3) each repair replica detects and repairs failures from any component represented in the Management layer, including the other repair replicas' nodes.

The fact that the replication is managed at the level of the component model is essential regarding the self-healing property of the repair service. In a previous design of our repair service [7], replication was applied at the level of the overall management system (any request made to the management system was replicated to another management system), the component-based internal organization of the management system being ignored by the replication service. While this design has made the repair service fault-tolerant, it has not allowed to provide a self healing property in a convenient way.

4.3 Self-Healing Repair Algorithm

This section describes the repair service algorithm based on the component features fulfilling the requirements previously exposed. In this algorithm, the expression *checkpoint-reference* (resp. *management-reference*) refers to a reference of a Checkpoint component (resp. *Management* component).

The main steps of the general algorithm are detailed in the following.

Algorithm 1 Global algorithm

```
Requires: An architecture-based management system satisfying the requirements exposed in this paper.
Ensures: The overall system is repaired in case of a node failure.
1: Analyse the failure and build a repair plan
2: Clean up the global system (remove failed components)
3: Execute the repair plan (substitute the failed components by newly ones)
```

4.3.1 Analysing the failure and building a repair plan.

This step (see Alg. 2) consists in analysing the failure and subsequently returning a repair plan, according to a repair policy. It uses the Checkpoint Layer to introspect the management state of the application prior to the failure, and inserts the references of the components to repair in the repair plan.

Algorithm 2 Analyse the failure & build a repair plan

```
Requires: $FailedNode$ : The checkpoint-reference of the component representing the failed node.
Ensures: $FailedCmps$: The management-references of the failed components. $RepairPlan$: A repair plan composed of the checkpoint-references of the components to repair (the default policy defines the components to repair as those which were running on the failed node)
1: for all $cmp$ in $FailedNode.getSubComponents()$ do
2:     $FailedCmps.addCmp(managementReference(\text{cmp}))$
3:     $RepairPlan.addCmp(\text{cmp})$
4: end for
```

4.3.2 Cleaning up the global system.

Cleaning up the global system (see Alg. 3) means locating and removing all references from a surviving component to a failed component in the Management layer. This is done by using the reconfiguration meta-operations of the component model. Because these operations are specialized for each type of legacy element, the necessary actions will be automatically performed at the Legacy layer (e.g., closing a TCP connection).

4.3.3 Execute the repair plan.

The checkpoint-references of the components to repair have been registered in the repair plan. Each of these components can be introspected to get its management state in order to substitute it by a new one having the same management state (see Alg. 4).

Algorithm 3 Clean up the global system

```
Requires: $FailedCmps$ : The management-references of the failed components.
Ensures: All relationships (binding, containment) involving a failed component are closed and removed.
1: for all $cmp$ in $FailedCmps$ do
2:     {for each alive component, remove a failed binding}
3:     for all $itf$ in $cmp.getServerInterfaces()$ do
4:         for all $clientItf$ in $itf.getBindingSources()$ do
5:             if $clientItf.owner()$ not in $FailedCmp$ then
6:                 $clientItf.unbind()$
7:             end if
8:         end for
9:     end for
       {for each alive component, remove a failed child}
10:    for all $parentCmp$ in $cmp.getParents()$ do
11:        if $parentCmp$ not in $FailedCmp$ then
12:            $parentCmp.removeSubComponent(cmp)$
13:        end if
14:    end for
       {for each alive component, remove a failed parent}
15:    for all $cmp$ in $FailedCmp$ do
16:        for all $subCmp$ in $cmp.getSubComponents()$ do
17:            if $subCmp$ not in $FailedCmp$ then
18:                $subCmp.removeParent(cmp)$
19:            end if
20:        end for
21:    end for
22: end for
```

Algorithm 4 Execute the repair plan

```
Requires: $RepairPlan$ : a repair plan as returned by Algorithm 2, $FailedNode$ : the checkpoint-reference of the component representing the failed node.
Ensures: The failed components are replaced by newly ones having the same management state
1: $newNode = NodeAllocator.replaceNode(FailedNode)$
2: for all $cmp$ in $RepairPlan$ do
3:     {Create an equivalent component (same attributes, interfaces, relationships and life cycle state) and deploy it on $newNode$}
4: end for
```

5. EXPERIMENTAL RESULTS

The ideas presented in this paper result from our experience with building and using the Jade architecture-based management system. Using this experience we assess the fact that a component model fulfilling the identified requirement allows building a self-repair service. This service has been validated both from a qualitative and a quantitative point of view. The rest of this section gives details on both Jade and the experiments. Then we evaluate the genericity of the requirements. On the basis of a self-repair service prototype applied to a J2ee system, we evaluate the performance overhead induced by the requirements and the gain in availability on such a legacy system.

5.1 Context

In the JADE architecture-based management system, we used the FRACTAL [8] reflective, Java-based component model, which is intended for the construction of dynamically configurable and monitorable systems.

The FRACTAL components have a reflective structure that is organized into membrane and content. The membrane of a component defines its abstractions and its meta-level methods, organized in specializable *controllers* and providing the introspection and reconfiguration operations as exposed in Section 3.

The components were enhanced with a meta-data checkpointing facility, and organized to support replication capabilities such as presented in Section 4. More precisely, any component can be tagged with the *replicated* and/or *checkpointed* property, which are provided in a generic fashion by the component programmer's point of view. Replicated components are managed through a semi-active replication strategy which deals with side-effects of server component (typically replicated component with both server and client interfaces). The protocol is asymmetric in the sense that one replica has the status of leader, and clients interact only with a leader replica. When the leader receives a request, it broadcasts it to the follower replicas using a FIFO broadcast protocol (all replicas are members of a group). Atomic broadcast is not required since there is at that moment only one sender in a given group (the leader). Then both leader and follower process the request and only the leader sends its response to the client (follower requests are just locally logged). If replicated components own a client interface, since requests are processed by all replicas, the component's interface to which the replicated component is bound will receive replicated requests. Since interaction is not necessarily idempotent, detection and filtering of duplicate requests is performed.

5.2 Genericity supported by the requirements

We recall that one of the JADE's objectives is to manage a variety of legacy software systems, regardless of their specific interface and underlying implementation. In order to evaluate this, we applied the JADE's management functions over two different legacy systems: a J2EE clustered web server and a JMS message server. To this end, we provide two specialized implementations of the management API described in Section 3, one for wrapping each legacy system. Experience on J2EE used the RUBiS benchmark, which implements an auction site [3]. RUBiS defines several web interactions (e.g., registering new users, browsing, buying or selling items); and it provides a benchmarking tool that emulates web client behaviors and generates a tunable workload (300 web clients). We used the RUBiS 1.4.2 version of the multi-tier J2EE application running on several middleware platforms: Apache 1.3.29 as a web server, Jakarta Tomcat 3.3.2 as an enterprise server and MySQL 4.0.17 as a database server. Experience on JMS used JORAM 4.3.12 as a JMS server. Due to the lack of JMS open source benchmark, we used a micro-benchmark composed of an arbitrary set of queues and topics. A synthetic workload is injected through a set of message producers and consumers.

Table 1 gives the code size of JADE's generic and specific sub-systems. It provides a rough measure of the code factoring obtained thanks to the generic approach followed in JADE. Indeed, taking into account a new administered legacy system in JADE would require to implement a JADE wrapper that consists of, in average, 360 lines of Java code and a FRACTAL configuration file of 20 lines (i.e. FRACTAL ADL). On the other hand, with an ad-hoc (i.e. non-generic) approach, taking into account a new legacy system would require to re-implement a new version of a repair service (with a total code size around 4750 lines of Java code).

			# Java lines	# ADL lines
Generic code		Deployment Service	3505	1690
		Checkpoint layer	6630	–
		Replication layer	4567	832
		Self-Repair service	4750	430
		Total	***19452***	***2952***
Specific code	J2EE	Rubis app. - Web	150	11
		Rubis app. - Servlets	150	11
		Rubis app. - Database	150	11
		Total	***450***	***33***
		Apache Web server	800	16
		Tomcat Servlet container	550	12
		MySQL SGBD	760	40
		Total	***2110***	***68***
	JMS	JORAM server	368	51
		JNDI	134	12
		JMS Queue	253	16
		JMS topic	297	16
		Total	***1052***	***95***

Table 1: Generic code vs. specific code

	Human admin.	JADE admin.
Throughput	12 req./s	12 req./s
Resp. time	87 ms	89 ms
Mem. usage	17.5 %	20.1 %

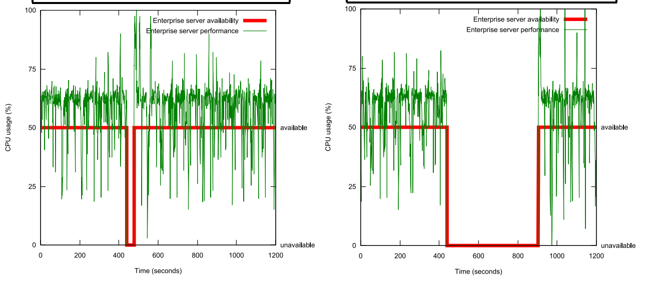
Figure 6: Throughput, response time and memory usage of RUBiS with and without Jade.

5.3 Performance overhead induced by the requirements

In order to measure the possible performance overhead induced by JADE, we compared two executions of the same multi-tier J2EE system: when it is run over JADE and when it is run without JADE. Experiment were performed on the Linux kernel running x86-compatible machines, with 1 GB RAM and 1800 MHz, connected via a 100 Mb/s Ethernet LAN to form a cluster. During the experiments, the managed application has been submitted to a medium workload without any failure so that its execution under the control of JADE induced no dynamic reconfiguration. The results (see Fig. 6) show no significant overhead in terms of application response times and throughput. We can notice a slight memory overhead (20.1% vs. 17.5%) that can be linked with the creation of internal software components by JADE. However, JADE does not induce a perceptible overhead on CPU usage; this is due to the fact that JADE does not intercept application communications but only configuration/management operations.

nb. failed req.	1900
MTTR	43 s
Availability	0.96

nb. failed req.	18700
MTTR	464 s
Availability	0.72



(a) With JADE (b) Without JADE

Figure 7: Enterprise server behavior in the presence of failures.

5.4 Guarantying service continuity

Figure 7 (a) & (b) compares the availability of the RUBiS application, in the presence of node failures, when the overall infrastructure is managed by a human administrator versus JADE. Under human administration (c.f. Fig. 7 (b)), when a failure occurs on the system (at time 440 seconds), the auction site becomes unavailable and all new client requests result in an HTTP error. To handle the failure, the operator must react as follows: (1) first, failure must be detected; (2) the operator needs to know in detail the system architecture; (3) based on this knowledge, he should diagnose the node failure and infer what reconfiguration is needed (which software need to be redeployed and which software configuration need to be updated); (4) the operator has to log on remote machines and manually perform these reconfiguration tasks. During this time, client requests (18700 requests until the end of the recovery) cause an HTTP error. On the other hand, when JADE is used (see Fig. 7 (a)), only 1900 failed client requests have been submitted by recovery-time. In our experiment, the human administrator was an expert and was waiting for a failure during all the experiment (that is why the $MTTR$ is so low). Oppenheimer et al. have shown in [19] that in real conditions, operator error is the largest contributor to $MTTR$, which could reach up to 9 hours.

6. RELATED WORK

JMX [18] is a de facto standard for administering Java applications such as J2EE application servers. It defines a three-layered management architecture, in which the lower layer is composed of components, called MBeans, representing the Java objects to manage. The Mbeans provide instrumentation interfaces, allowing mainly updating the configuration attributes and getting statistics and performance information. The JSR77 standard defines the MBeans attributes according to a hierarchical information model of J2EE managed objects. According to the requirements presented in this paper, the main limitation of JMX is that it does not expose the dependencies (in terms of bindings and containment) between the managed objects. Building sophisticated management services requiring the knowledge of the current overall architecture of a J2EE server (e.g., current number of replicates) is not possible, unless this architecture is immutable.

Models for distributed applications management have been considered by the DMTF (Distributed Management Task Force) [1] through the definition of the CIM (Common Information Model) / WBEM (Web-Based Enterprise Management). CIM is an object-oriented information model that provides a conceptual view of physical and logical system components through the classical abstractions provided by object models. A managed element is represented by an object which can have different kinds of relations (associations) with other objects. However, no uniform management interface is defined (management operations are provided as specific object methods).

SMARTFROG [15] is a framework for the management of configuration-driven systems. It defines a system as a collection of software components with certain properties (i.e., attributes). The framework provides mechanisms for describing these component collections and for deploying and managing their life cycle. The SMARTFROG component model defines the interfaces that a software component should implement (or that can be provided by a management adapter). These are based on three main abstractions: attributes, life cycle and containment. A limited notion of binding between components is however provided through the concept of link, which allows to manage a dependency between two components attributes values. The containment abstraction is defined as a way of providing a hierarchical naming scheme by which attributes may be referenced. It is not clear whether and how containment relationships can be introspected and manipulated dynamically. Moreover, while the life cycle operations are specializable, this is not the case for the other kinds of operations (dynamic attributes or containment manipulation). Finally, like in the previous component models, no non-functional property is provided.

Lira [10] is a lightweight infrastructure for managing dynamic reconfiguration that applies and extends the concepts of network management to distributed software systems. Individual managed entities are under the control of agents which perform basic reconfiguration functions on them. The main abstractions and operations provided by an agent are (1) a life cycle state machine, which is manipulated through an extensible and specializable set of predefined functions such as start/stop; (2) a setter/getter interface allowing to attributes to be manipulated; and (3) a *call* operation that may invoke any specific method defined by the agent. There are no explicit containment and binding notions. The Lira infrastructure was experienced with the automatic failure-repair of Siena [9], a distributed, content-based, publish-subscribe event notification service. It is not clear how the repair manager can get, after the failure, the necessary information allowing the system to be repaired in a generic manner.

[12] proposes an infrastructure for creating architecture-based self-healing systems. Event-based software architectures are targeted, because the managed software elements are loosely coupled (an element can be replaced without impacting the other elements). They use a component- and message-based runtime infrastructure to represent and manage the running legacy system. The main provided abstractions are based on the notions of component, ports (i.e., interfaces), connectors (i.e., bindings), and messages (representing requests or notifications). Dynamic addition or removal of components is supported. The aspects related to the reliability of the components are presented as a future

work.

[11] proposes a self-repair management framework based on architectural styles. An architectural style defines a set of formally specified constraints over an architecture (a constraint violation being a cause for inducing a repair), as well as how to carry out repair in terms of high-level architectural operators. A style-specific translation service maps the high-level architecture operations into lower-level systems operations acting on runtime components. Architectural styles can be used complementary to the principles of architecture-based management framework exposed in this paper, as a way of (1) constraining the dynamic evolution of an architecture; and (2) specializing some high level management functions.

Architectural styles are also considered in the Plastik infrastructure [5], where the approach consist in building general invariants into the specification of a component-based system and to accept any change as long as the invariants are not violated. The component model is based on OpenCom, whose main abstractions are: interfaces, bindings, containment, attributes and the notion of component framework corresponding to tightly coupled clusters of components cooperating toward a common functional goal. The OpenCom runtime supports reflective meta-models which provides basic introspection and reconfiguration facilities and allows specializing these operations. The introspection facilities maintain an architectural view of an application, which is used to perform reconfiguration operations. This approach, as well as [11], conforms to [21], which exposes architectural style requirements for building self-healing systems. These requirements are refined by those exposed in this paper which more specifically consider management systems based on runtime components.

[14] proposes a component model including self-organising properties. Components are associated with constraints that define their behavior according to the architectural evolution of the global system. Components are containers of provided and required interfaces, which can be connected through bindings. Hierarchical component composition is supported. The basic management operations are adding/removing a component, as well as adding/removing a binding. Specialization of these operations on a component-basis is supported. Moreover, each component holds a checkpoint of the current architectural state of the *global* system, allowing it to adjust its current configuration in accordance with this state. The checkpoint views are managed through a group protocol, which tolerates the failure of individual components. The component model proposed by [14] partially overlaps the requirements expressed in this paper. However, abstractions such as attributes or life cycle state are not mentioned.

Finally, [6] considers the use of reflective middleware to develop self-managing systems as a challenging research direction. They more specifically target platforms where both middleware and applications are built with reflective autonomic components. Like us, they put a particular emphasis on the role of components and reflection and their use to provide general introspection and adaptation capabilities.

7. CONCLUSION

Components provide a way of building architecture-based management systems, which exhibit the software architecture of a managed application and provide a way of adapting it to changing environments. More precisely, the components are used to represent the architecture of a managed application as an assembly of components providing a uniform management interface. They may also be used for implementing management services.

However, based on our experience in building an autonomic repair service in the context such systems, we observed that the properties expected from the component model are strong, and seldom entirely satisfied by the existing component models. An autonomic repair service must not only repair the applications that it manages but also demonstrate a self-healing behavior meaning that it is able to repair itself. The required properties are summarized as follows:

- To build an autonomic repair service, the component model should (1) provide five main runtime abstractions: attributes, interfaces, life cycle state, binding and containment; and (2) provide a way to manipulate these abstractions through *specializable* meta-operations (e.g., addSubComponent, bind, etc.).
- To enhance the repair service with self-healing behavior, the components have to support two non-functional properties: a *checkpointed* property, allowing a component's meta-data to be checkpointed (e.g., containments, bindings, etc.), and a *replicated* property allowing components to be replicated.

Our experience has showed the importance of non-fonctional aspects of the component model such as meta-data checkpointing and replication.

The soundness of our findings has been validated through several large experiments on successful middleware platforms. One is a clustered J2EE Web server and the other is a Jms message server.

Acknowledgments.

We would like to acknowledge Prof. Sacha Krakowiak and Prof. Olivier Gruber from University Grenoble 1 for their help in reviewing this paper.

8. REFERENCES

[1] DMTF (Distributed Management Task Force). http://www.dmtf.org/home.

[2] EGA (Enterprise Grid Alliance), Reference Model and Use Cases v1.5, 2006. http://www.gridalliance.org.

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

[4] A. Avizienis, J.C. Laprie, B. Randell, and C. Landwehr. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Transactions on Dependable and Secure Computing*, 01(1), 2004.

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

[6] G.S. Blair, G. Coulson, and P. Grace. Research directions in reflective middleware: the Lancaster

experience. In *Proceedings of the 3rd Workshop on Adaptive and Reflective Middleware (ARM'04)*, New York, NY, USA, 2004. ACM Press.

[7] S. Bouchenak, F. Boyer, D. Hagimont, and S. Krakowiak. Architecture-Based Autonomous Repair Management: An Application to J2EE Clusters. In *24th IEEE Symposium on Reliable Distributed Systems (SRDS-2005)*, Orlando, FL, October 2005.

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

[9] A. Carzaniga, M. J. Rutherford, and A.L. Wolf. A Routing Scheme for Content-Based Networking. In *Proceedings of IEEE INFOCOM 2004*, Hong Kong, China, March 2004.

[10] M. Castaldi, A. Carzaniga, P. Inverardi, and A. Wolf. A Lightweight Infrastructure for Reconfiguring Applications. In Springer-Verlag, editor, *11th International Workshop on Software Configuration Management*, Berlin, 2003.

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

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

[13] 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), October 2004.

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

[15] P. Goldsack, J. Guijarro, A. Lain, G. Mecheneau, P. Murray, and P. Toft. Configuration and Automatic Ignition of Distributed Applications. In *HP OVUA 2003 - HP OpenView University Association*, 2003.

[16] G. Kiczales and J. Des Rivieres. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, USA, 1991.

[17] A.I. Kistijantoro, G. Morgan, S.K. Shrivastava, and M.C. Little. Component Replication in Distributed Systems: A Case Study Using Enterprise Java Beans. In *22th IEEE Symposium on Reliable Distributed Systems (SRDS-2003)*, 2003.

[18] Sun Microsystems. Java(TM) Management Extensions (JMX TM). `http://java.sun.com/javase/technologies/core/mntr-mgmt/javamanagement/`.

[19] D. Oppenheimer, A. Ganapathi, and D. Patterson. Why do Internet services fail, and what can be done about it? In *USENIX Symposium on Internet Technologies and Systems (USITS'03)*, 2003.

[20] JOnAS Project. Java Open Application Server. http://jonas.objectweb.org.

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

[22] SmartFrog. SmartFrog: Smart Framework for Object Groups. HP Labs. `http://www.hpl.hp.com/research/smartfrog/`.

[23] BEA Systems. Achieving Scalability and High Availability for E-Business, January 2004. `http://dev2dev.bea.com/pub/a/2004/01/WLS\_81\_Clustering.html`.

[24] C. Szyperski. *Component software: beyond object oriented programming*. ACM Press Addison-Wesley Publishing Co., New York, NY, USA, 1998.