Architecture-Based Autonomous Repair Management: An Application to J2EE Clusters

Sara Bouchenak[1], Fabienne Boyer[1], Daniel Hagimont[2], Sacha Krakowiak[1], Adrian Mos[2], Noel de Palma[3], Vivien Quema[3], Jean-Bernard Stefani[2]

[1] Joseph Fourier University Grenoble, France
[2] INRIA Grenoble, France
[3] INPG Grenoble, France

Abstract

This paper presents a component-based architecture for autonomous repair management in distributed systems, and a prototype implementation of this architecture, called JADE, *which provides repair management for J2EE application server clusters. The* JADE *architecture features three major elements, which we believe to be of wide relevance for the construction of autonomic distributed systems: (1) a dynamically configurable, component-based structure that exploits the reflective features of the* FRACTAL *component model; (2) an explicit and configurable feedback control loop structure, that manifests the relationship between managed system and repair management functions; (3) an original replication structure for the management subsystem itself, which makes it fault-tolerant and self-healing.*

1 Introduction

Autonomic computing [21], which aims at the construction of self-managing and self-adapting computer systems, has emerged as an important research agenda in the face of the ever-increasing complexity and pervasiveness of networked computer systems. Following [31], we believe an important part of this agenda lies in the elicitation of architectural principles and design patterns, as well as software engineering techniques for the construction of autonomic systems. We believe in particular that we need to develop a principled approach to designing and architecting autonomous management systems, which remains as independent as possible from the implementation details of managed systems. As a contribution to this goal, we present in this paper the design of a self-healing failure repair management system, and the application of this design to the construction of an autonomous repair management system for J2EE clusters.

We call *repair management* an instance of failure recovery management, whose main goal is to restore a managed system, after the occurrence of a failure, to an active state satisfying a given level of availability, according to a given policy. For instance, a simple repair policy can be to bring back the failed system to a known configuration which existed prior to failure. Typically, repair management can be used in complement of classical fault-tolerance mechanisms to ensure a managed system satisfies an agreed level of availability: for instance, a cluster system that is tolerant of f node failures, can become only $f-1$ tolerant if a node fails; repair management can bring the system back to an f-tolerant functioning regime. The repair management system presented in this paper is *self-healing* in that it is fault-tolerant, and deals with failures occurring in the repair management system itself, without any human intervention.

Our autonomous repair management design contains three main system architecture and software engineering elements, which we believe to be of wide relevance for the construction of self-managing and self-adapting systems, and which constitute the main contributions of the paper: (1) a dynamically configurable component-based architecture and its supporting software infrastructure; (2) an explicit (and configurable) feedback control loop structure, that manifests the relationship between managed system and repair management functions, and which maintains a causally connected representation of the managed system; (3) an original replication structure for the management system, which makes it fault-tolerant and self-healing.

We have applied our repair management design to build a prototype repair management system for J2EE application server clusters, called JADE. Apart from the fact that Web application servers constitute an important and representative segment of distributed computing, there are several reasons for this choice of application area and experimental setting. First, performability management in application servers still remains an open issue, witness several recent papers that deal with this subject, such as [9, 16]. Our repair management system improves on the state of the art in this area by demonstrating how availability management can be made entirely automatic, even in presence of failures in the management sub-system itself. Second, a J2EE application server cluster constitutes a distributed system which has a non-trivial complexity, involving several interacting tiers, and combining different legacy middleware technologies (web servers, EJB servers, databases). We demonstrate with the JADE prototype that our design for repair management can be applied to non-trivial legacy systems, with essentially no modification of the legacy code. Third, the size of J2EE cluster remains limited, allowing for practical experiments, but scaling issues are already apparent when using group communication protocols such as atomic broadcast (see e.g. [1] for a discussion). We demonstrate with the JADE prototype that our design for repair management can be scaled to cluster size distributed systems.

The paper is organized as follows. Section 2 presents the main requirements and underlying assumptions on the environment for our repair management design. Section 3 presents the FRACTAL component model, which constitutes the basis for our design. Section 4 presents the overall feedback control loop the repair management system realizes. Section 5 describes the replication structure we use to make the repair management system fault-tolerant and self-healing. Section 6 describes a prototype implementation of our design called JADE, which applies it to the construction of self-repairable J2EE application server clusters. Section 8 discusses related works. Section 9 concludes the paper with a discussion of future research.

2 Requirements

We gather in this section the main requirements and underlying assumptions for our design. We distinguish between functional requirements, which mandate the presence of specific capabilities, and non-functional ones, which require that certain properties be achieved in the design and its implementation.

The main *functional requirements* for our repair management system are given below.

Failure modes: the repair management system should deal with silent hardware (machine crashes) and software failures. An underlying assumption is that the interconnection network between the supporting machines is a mesh, has known upper bounds on communication delays, and does not fail[1].

Basic repair policy: the repair management system should support at least a basic repair policy consisting in updating the failed managed system to a configuration that conforms to the same architecture description than the configuration in place prior to the occurrence of failure.

Fault-tolerance: the repair management system should be fault-tolerant with respect to the failure (crash) of machines on which it executes.

[1]These are simplifying assumptions, in line with the high-performance clustering environments we target with the JADE prototype. The overall design would not be significantly impacted if these assumptions were invalidated, however, since all communication paths in our component-based design can be reified as components.

Self-repair: the repair management system should be self-healing in the sense that it should deal with failures occurring repair management system itself. In other words, repair management should function also to repair itself.[2]

Dynamic system configuration: the repair management system should be dynamically deployable. In particular, it should be possible to set up repair management on an already running managed system, and it should be possible to dynamically modify the different elements of the repair management system (e.g. its management policy, its repair algorithms, its monitoring structure, etc).

Support for ad-hoc administration: the repair management system should allow human administrators to monitor the activity of the overall system (managed J2EE system as well as repair management system), and to manually effect repairs, in replacement of the standard repair management behavior. A human administrator should in addition be allowed to dynamically set up a management console on an arbitrary node (i.e. a management console should not be tied to a specific node in the system and should be dynamically deployable, as per the dyanmic configuration requirement above).

Support for legacy systems: the repair management system should be applicable to legacy systems (possibly with reduced requirements)[3]

The main *non-functional requirements* for our design are given below.

Strict separation between mechanism and policy: the repair management system should enforce a strict separation between mechanism and policy, with the goal to allow the execution of different management policies, possibly at different levels of abstraction. In particular, it should allow for different configuration and deployment policies, from the exact specification of selected components and locations at which they should be deployed, to the indirect specification of selected components and their locations through high-level management goals and constraints.[4]

Minimal interference: the operations of the management sub-system should interfere minimally with the behavior of the managed system. Thus:the repair management system should preserve the correct operation of the managed system, and the performance interference between the nominal behavior of the system and its behavior under management should be minimal.

3 Component basis

Our repair management system is built using the FRACTAL reflective component model [7]. In this section, we present briefly the FRACTAL model, and how we use it.

3.1 The FRACTAL component model

The FRACTAL component model is a general component model which is intended to implement, deploy, monitor, and dynamically configure, complex software systems, including in particular operating systems and middleware. This motivates the main features of the model: composite components (to have a uniform view of applications at various levels of abstraction), introspection capabilities (to monitor, and control the execution of a running system), and re-configuration capabilities (to deploy, and dynamically configure a system). A FRACTAL component is a run-time entity that is encapsulated, and that has a distinct identity. A component has one or more interfaces, and the number of its interfaces may vary during its execution. An interface is an access point

[2]Note that this is a different requirement than the fault tolerance requirement above.

[3]A legacy application usually appears as a "black box" providing immutable interfaces for activation and monitoring; its internal structure is not accessible.

[4]For instance, one can have different levels of declarativity in the specification of deployment and configurations: the selected components and the location at which they should be deployed are explicitly specified; the components and locations are not explicitly specified, but the administrator may specify preferences;the specification is in terms of higher-level goals, e.g. select the components and the locations for maximal availability or maximal performance.

to a component, that supports a finite set of methods. Interfaces can be of two kinds: server interfaces, which correspond to access points accepting incoming method calls, and client interfaces, which correspond to access points supporting outgoing method calls. The signatures of both kinds of interface can be described by a standard Java interface declaration, with an additional role indication (server or client). A FRACTAL component can be composite, i.e. it may contain other components.

Communication between FRACTAL components is only possible if their interfaces are bound. FRACTAL supports both primitive bindings and composite bindings. A *primitive binding* is a binding between one client interface and one server interface in the same address space. A *composite binding* is a FRACTAL component that embodies a communication path between an arbitrary number of component interfaces. These bindings are built out of a set of primitive bindings and binding components (stubs, skeletons, adapters, etc). Note that, except for primitive bindings, there is no predefined set of bindings in FRACTAL. In fact bindings can be built explicitly by composition, just as other components. The FRACTAL model thus provides two mechanisms to define the architecture of an application: bindings between component interfaces, and encapsulation of components in a composite.

The above features (hierarchical components, explicit bindings between components, strict separation between component interfaces and component implementation) are relatively classical. The originality of the FRACTAL model lies in its open reflective features. In order to allow for well scoped dynamic reconfiguration, components in FRACTAL can be endowed with controllers, which provide a meta-level access to a component internals, allowing for component introspection and the control of component behaviour. In contrast to other reflective component models such as OpenCOM [13], FRACTAL allows for arbitrary forms of control and component reflection.

At the lowest level of control, a FRACTAL component is a black box, that does not provide any introspection capability. Such components, called base components, are similar to plain objects in an object-oriented language. Their explicit inclusion in the FRACTAL model facilitates the integration of legacy software.

At the next level of control, a FRACTAL component provides a Component interface, similar to the IUnknown in the COM model, that allows one to discover all its external (client and server) interfaces. At upper levels of control, a FRACTAL component exposes (part of) its internal structure. The structure of a FRACTAL component then can be seen as comprising two parts: a *membrane*, which provides interfaces (called *control interfaces*) to introspect and reconfigure its internal features, and a *content*, which consists in a finite set of components. The membrane of a component is typically composed of several controllers. A controller implements control interfaces and can typically superpose a control behavior to the behavior of the component's sub-components, including suspending, checkpointing and resuming activities of these sub-components.A controller can also play the role of an interceptor,used to export the external interface of a subcomponent as an external interface of the parent component, and to intercept the oncoming and outgoing method calls of an exported interface. The FRACTAL model allows for arbitrary (including user defined) classes of controller and interceptor objects. It specifies, however,several useful forms of controllers, which can be combined and extended to yield components with different reflective features, including the following:

- **Attribute controller:** An attribute is a configurable property of a component. This controller supports an interface to expose getter and setter methods for its attributes.

- **Binding controller:** supports an interface to allow binding and unbinding its client interfaces to server interfaces by means of primitive bindings.

- **Content controller:** supports an interface to list, add and remove subcomponents in its contents.

- **Life-cycle controller:** This controller allows an explicit control over a component execution. Its associated interface includes methods to start and stop the execution of the component.

3.2 Using components and software architecture

We use the component model in three main ways: for obtaining a dynamically reconfigurable structure, for instrumenting the managed system, and for building a causally connected system representation.The construction of a system with FRACTAL components yields a dynamically reconfigurable system, where a component is a unit of reconfiguration. We exploit this for the dynamic deployment of the repair management system. Each component of the repair management system has three controllers (binding, content, and lifecycle) that allow updating the configuration of the management system.Moreover, instrumentation of the managed system can be done using a set of controllers behind which legacy software is wrapped. These controllers can provide the same instrumenting capability as a dynamic aspect weaver (e.g., method interception, component state intercession), in an explicit system architecture. In particular, controllers provide us a way to build behavior monitoring and tracing facilities adapted to each component, to control the lifecycle of components, and to control the way component interfaces are bound and interact. More generally, component controllers can implement sensors and actuators required by a management control loop.Central to the operation of the repair management system, is a representation of the managed system, called the *system representation*. A causal connection is maintained between the system representation and the managed system, which means that relevant state changes occurring in the system are mirrored in the system model, and, conversely, that any changes on the system representation are reflected in the actual evolution of the system. The system representation consists in a software architecture description of the system, expressed in terms of the component model, exhibiting bindings between components, and containment reltionships. Components are run-time entities in FRACTAL, so the software architecture description of the managed system in our repair management design mirrors the actual run-time structure. The software architecture description is generated directly from a specification written in the FRACTAL architecture description language (ADL).

4 Overall architecture

Our repair management architecture can be described literally as a software architecture conforming to the FRACTAL component model. An instance of our repair management architecture constitutes a management domain, i.e. a set of components under a single repair management authority. Our architecture can extend to the case of hierarchically organized management domains, but we do not describe this extension in this paper. To present our repair management architecture we first define the components that constitute a repair management domain. We then present the overall structure of the repair management feedback control loop. The replication structure for the repair management system, which ensures it self-healing property, is described in the next section.

4.1 Repair management domain

The scope of a repair management domain, in our design, is defined by a set of components, called *nodes*, together with their sub-components. A node corresponds to an abstraction of a physical computer (typically in a PC cluster). Sub-components of a node correspond to the different software components executing in the node. The set of nodes in a repair management domain, and the components they support, can vary dynamically over time, due to failures, and to the introduction of new nodes. The introduction of a new node in a management domain is the responsibility of the *node manager* component. The node manager provides two basic operations: an operation to request the introduction of a new node, and an operation to collect a failed node. The introduction of a new node can be understood, from the viewpoint of other components in the repair management system, as the instantiation of a new node component in its initial state.

A node component provides operations for deploying and configuring components executing on the node. The state of a node component is determined by its set of subcomponents, together with their established intra, and inter-node bindings. In its initial state, the set of components executing on a node is empty. Note that components executing in a node correspond to sub-components, in FRACTAL terms, of the node component. Operations provided by a node component include the addition and removal of components to and from a node. For instance, in the JADE system, which applies our repair management architecture to J2EE application server clusters, components which can be added to, or removed from, a node include:

- Middleware components that comprise the different tiers in a J2EE server.
- Application components that comprise Web service applications in a J2EE server.
- Node failure sensor components, which enable the remote monitoring of a node status (active or failed).
- Components that form the repair management sub-system per se, described in the next subsection.

In addition, a node provides basic lifecycle operations to stop a running node and to resume execution of a node. A node also provides operations to modify the code (e.g. version) of the components it contains. Middleware and application components appearing in the different tiers of the managed J2EE server are actually *legacy* components in the JADE prototype. They are wrapped as FRACTAL components exhibiting the attribute controller interface and the binding controller interface.

Some remarks are in order. First, note that instantiating a node implies bootstrapping the underlying machine with some remote communication capabilities, and can imply passing the node manager appropriate values (e.g. software packages) in a request for introduction of a new node. Second, note that a repair management domain explicitly comprises all the components that form the repair management system itself, meaning that repair management system itself is subject to its repair management policy. Third, note that the addition and removal of components in node correspond to the basic operations of the FRACTAL content controller interface, while the lifecycle operations of a node correspond to the FRACTAL lifecycle controller interface.

4.2 Repair management control loop

The overall structure of the repair management system takes the form of a system feedback control loop, as illustrated in Figure 1. We describe each element of the loop.

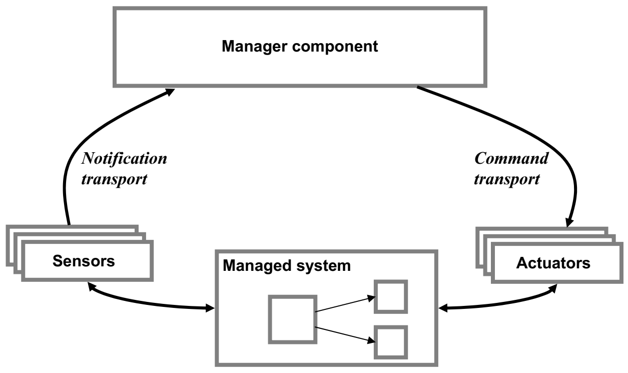

Figure 1. Management control loop

Sensors. Basic observations handled by the repair management system comprise: component state changes, node resource usage observations, and component failure observations (failure of a node component corresponds of course to the crash of the corresponding physical machine, in case of a direct mapping node/machine). Node failure observations are the result of a failure detection protocol that is run between monitored nodes and the monitoring node. The failure detection protocol involves components residing on each monitored node, components residing on the monitoring node, and bindings between them. The failure of a node results in a `failed-node` event being generated at the monitoring node.

Actuators. The repair management system exercises the following basic actions[5] on managed components (including nodes):

- Lifecycle actions: to stop and resume execution of components in a node, and to stop and resume execution of binding components (including bindings between nodes). These actions are available as operations on the life-cycle controller interfaces of these components (including for wrapped legacy components).
- Configuration actions: to add or remove a component from a node, and the creation or removal of a binding component (including bindings between nodes); to modify the code of running components. The former actions are available as operations on the content controller interface of each node or binding component. The latter actions are available as operations on the content controller interface of the selected components.

Transport. Bindings that bind sensors, actuators and the Manager component, are referred to, collectively, as the transport subsystem. Notification of failures follow an asynchronous operation semantics, but are subject to timing and safety guarantees (e.g. notifications of failure should take place within a specified delay bound; notification loss should be below a given threshold). Commands issued by the Manager component (typically, configuration actions) obey a synchronous operation at most once semantics, under timing constraints.

Manager. The Manager component implements the analysis and decision stage of the repair management control loop. An example manager component is illustrated in Figure 4. The Manager component contains several subcomponents: a set of policy components (configuration manager, console, and repair manager), and a system representation component. The system representation maintains a causally connected representation of the managed system. The configuration manager enforces policies for installing components on nodes (e.g., deployment policies for J2EE tiers and for applications). The repair manager enforces the repair management policy per se. The console provides a monitoring console and a script interpreter that allow a human administrator to monitor the state of the system, and to issue commands to modify the system state.

The system representation component maintains an abstract view of the system required for the operation of the policy components. The system representation corresponds to a software architecture description of the actual system, and the causal connection between the system model and the actual system is manifested by explicit bindings between managed components and (sub-components of) the Manager component. Causal connection is maintained as follows:

- On bootstrap of the repair management system, the system representation contains an architecture description of the system after initialization (including nodes that run the management sub-system with the components they contain).
- Policy components update the system model with the new nodes they have introduced (through requests to the node manager), if any, and the new components they have added to the active nodes in the domain.
- Policy components install or update bindings between each node they have installed or updated, and the Manager component.
- Upon receipt of a relevant state change from a sensor, the Manager component updates the system representation to reflect the new state of the system.

[5]Basic actions are actions which represent mechanistic, imperative commands, without intervening complex decision making in their realization – at least from the point of view of the management system. Actuators correspond to a level of mechanisms, as seen from the viewpoint of the management system. Policies are then built on top of these basic mechanisms.

- When a command is executed that modifies the state of the system, the Manager component transactionally updates the system representation to reflect the new state of the system.

The behavior of the repair manager is determined by the repair policy it enforces. Here is an example of a (simple) repair policy, implemented in the JADE system:

1. On receipt of a J2EE component failure, execute the following commands, in sequence: terminate bindings to the failed component; start a new (possibly the same) J2EE component replacing the previous one in the same node; re-establish failed bindings.

2. On receipt of a node failure, execute the following commands in sequence: terminate bindings to all the failed J2EE components which were supported by the node; request a new node to the node manager;[6] install a configuration isomorphic to the failed one on the selected node; re-establish all failed bindings to the new configuration.

3. If the node manager raises an exception, signal a node availability fault.

Several remarks are in order. The above policy implicitly considers stateless configurations, i.e. it assumes that there is no shared state information which is maintained across the different tiers and their associated components. More complex policies are required to deal properly with stateful component configurations. Also, the policy above is minimal in the sense that it considers all external requests (and their associated execution contexts) are equivalent. More complex policies are needed, for instance, when differentiation exists between requests, based on various factors such as session age or principals, e.g. to favor privileged customers in presence of resource shortages. In these cases the above policy would need to be adapted to check for resource capacity thresholds (e.g. number of available nodes) and possible establishment of admission control (e.g. to shed incoming un-privileged requests).Scenarios with differentiated services illustrate the need to consider the interaction of several system management function, in particular between failure management and performance management (where performance management is understood as the function guaranteeing a given level of performance to potentially different classes of customers). We leave this for future work.

5 Replication structure

The control loop structure presented in the previous subsection is sensitive to failures impacting the Manager component or the node running the Manager component (called the Manager node). We present in this section a replication structure which makes the Manager component tolerant of failures and which allows the repair management system to be self-healing in the sense that repair management policies enforce by the Manager component can be applied to the repair of the Manager component itself or of the node executing the Manager component. For simplicity, the description in this section considers only the case of machine failures.

The replication structure, illustrated in Figure 2, is obtained by: (1) actively duplicating the Manager component on different nodes, and (2) by ensuring that the System Representation contains a representation of the Manager nodes themselves (together with their internal configuration and bindings, as for other nodes). Self representation in the System Representation is easy to obtain by deriving the content of the System Representation component from an ADL description of the whole system, as explained in section 6. A simple active replication of the Manager component on different nodes is not sufficient, however, for there are subtle interplays between the different elements involved in the construction of the feedback loop described in the previous subsection. In particular, sensors for node failure detection are located on the Manager node. If the Manager node can fail,

[6]The node manager may itself enforce different node election policies. For instance, a node election policy can be given by the following failover scheme: allocate to each tier a given subset of the total set of nodes; partition each subset in two sets: active nodes in the tier and failover nodes in the tier. When an active node fails in a tier fails, choose an arbitrary failover node in the same tier as the elected node.

one must then replicate the node failure sensors as well. Another issue arises with the execution of the group communication protocol used for implementing the active replication of the Manager component: for reasons of efficiency and fault-tolerance, we make use of a uniform atomic broadcast protocol that uses a form of fixed sequencer (called Leader), with sequencer election in case of sequencer failure [15]; to ensure a consistent state of the Manager component replicas, the repair manager component must be able to assess the Leader status of its supporting node.

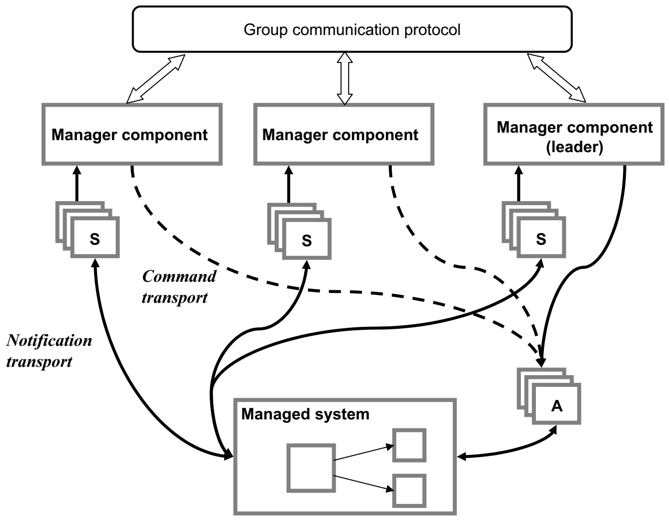

Figure 2. Replication structure

The active replication of the Manager components is thus realized according to the following scheme:

- The Manager component and the node failure detection sensors are replicated on different nodes, according to the required level of fault-tolerance (since we only consider silent failures with no possible network partition, f Manager node failures can be tolerated using $f + 1$ nodes.
- Each failure notification to, and each command from, a Manager component, is broadcast to all other Manager components in the system (Manager components form a single broadcast group in the system), using the uniform atomic broadcast protocol.
- Only the policy components of the Leader Manager node act on failure notifications or instructions from the Console. In particular, only the Leader policy components send requests to the node manager and issue configuration commands to managed nodes.
- Associations between notifications and resulting commands, as well as between console instructions and resulting commands, are assigned unique identifiers and journalized by policy components in each Manager. This allows, for example, the repair manager of a newly elected Leader to complete the reaction to a failure notification which has not been completed by the failed former Leader.
- Identifiers for notification/command pairs uniquely identify commands, and are sent as additional parameters of each command. Actuators discard commands they receive that bear the same identifier of a previously executed command. This takes care of the potential window of vulnerability that exists between the completion of a command and the failure of a Leader Manager node.

The uniform atomic broadcast protocol can be briefly described as follows. Uniform atomic broadcast allows broadcast of messages among a set of processes (Managers nodes in our case). Processes are assumed to communicate by asynchronous point-to-point message passing links (e.g. UDP or TCP in our cluster experiments).

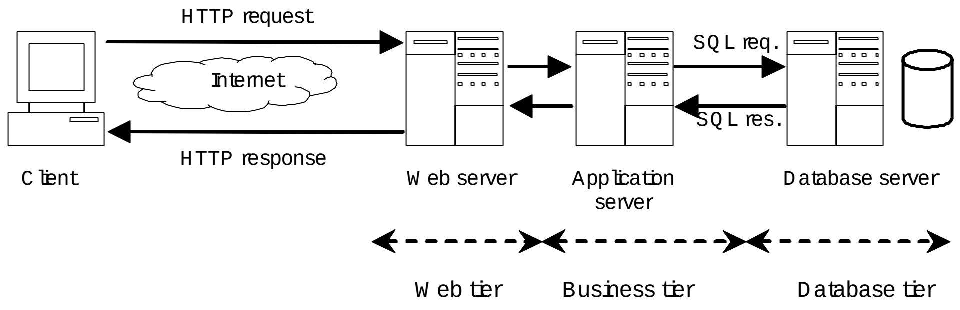

Figure 3. Architecture of J2EE applications

The operating principle of the protocol is as follows: two processes play a particular role: *Leader* and *Backup*. Any process p wishing to broadcast a message sends it to *Leader*, and keeps it in memory, until it receives its own message. If *Leader* crashes in the meantime, p sends the message to *Backup*. Upon receiving a message m to be broadcast, *Leader* serializes m by appending to m a unique sequence number $m.sn$, monotically increased by one after every use. *Leader* then sends m to *Backup* and waits for an acknowledgment from *Backup*. *Backup* broadcasts m to all processes belonging to the group (i.e. Manager added by the node allocator). In case of *Leader* crash, *Backup* becomes the new *Leader* and selects a new *Backup*. In case of *Backup* crash, *Leader* just selects a new *Backup*. To ensure a good scalability, message broadcast is not done using IP multicast, but using spanning trees and point-to-point communication links.

6 Implementation

The JADE prototype is an implementation of the repair management architecture described in sections 4 and 5. JADE targets specifically clusters of multi-tier J2EE application servers, however most of the JADE prototype can be exploited in other application areas. In this section, after introducing J2EE applications, we discuss two parts of the JADE prototype which are not covered by the architecture description in previous sections: the wrapping of legacy components in FRACTAL components, and the implementation of the system representation. We end this section with a brief presentation of a repair management scenario which has been experimented with JADE, a short note on the implementation of the communication infrastructure elements used in JADE, and a qualitative evaluation of the JADE prototype.

6.1. J2EE applications

The J2EE architecture defines a model for developing distributed multi-tiered applications. The application usually starts with requests from Web clients that flow through an HTTP server, then to an application server to execute the business logic of the application and dynamically generate Web pages, and finally to a database that stores the non-ephemeral data (see Figure 3).

6.2. Legacy component wrapping

We used FRACTAL to wrap the different software components which constitute a J2EE application. In our current prototype, these software bricks are namely: the Apache HTTP server, the Tomcat Servlet Container, the

Jonas EJB server and the MySQL database server. A FRACTAL component is associated with each of these bricks. When instantiated, this FRACTAL component is collocated with the wrapped software and it provides the control interfaces required for its management. It is implemented in Java and it translates all invocations on its control interfaces into adequate control operations on the legacy software.

For each component, we implement the following interfaces:

Attribute controller: The implementations of the attribute getter and setter methods provide a mapping of the component's attributes on the configurable attributes of the wrapped legacy software. The component's attributes and the legacy configurable attributes are not necessarily identical, i.e. the component's attributes can be much more abstract. For instance, the wrapping of an Apache server exposes as attributes most of the properties managed in the httpd.conf configuration file (User, Group, Port, ServerAdmin, etc.). The Apache wrapper uses the attributes values to generate the httpd.conf file when the server is started.

Life-cycle controller: The implementations of the life-cycle methods generally rely on system calls to run the script file (provided in the legacy software tools) associated with each operation.

Binding controller: In our J2EE application example, software tiers (Apache, Tomcat, Jonas, MySQL) are interconnected to form an architecture which implements a dynamic content web server. The definition of the tiers interconnection is defined in the legacy software configuration files (e.g. httpd.conf for Apache or server.xml for Tomcat). Our wrapping reifies these interconnections as FRACTAL bindings. For instance, in a J2EE application, the Apache and Tomcat tiers are interconnected using a mod_jk connection [29]. The definition of this connection is configured in the httpd.conf and server.xml configuration files (respectively for Apache and Tomcat tiers). Binding operations update these configuration files according to binding definitions.

Overall, wrapping legacy software bricks in FRACTAL components provides a uniform programmatic interface for their management in JADE. From there, higher abstraction level components can be defined for the autonomic administration of the whole clustered J2EE application.

6.3. System representation

The system representation provides a view of the runtime system architecture, which is isomorphic, introspectable and causally connected to runtime components. The system representation is built automatically from the ADL description of the managed system, and consists in a set of components, called *meta components*.

A meta component has the same interfaces, bindings, attributes and internal configuration (in terms of meta components) than the component it reifies (base level component). Controllers of a meta component implement the causal connection with the corresponding base level component. For instance, invoking a binding controller at the meta level to bind two meta components will invoke the corresponding binding controller at the base level.

Both the managed system and the management system are described with the FRACTAL architecture description language (ADL). Architecture descriptions are used by configuration managers to deploy all the system components at runtime. The FRACTAL ADL is in fact an extensible XML-based language, which allows different language modules and compiler components to be added and combined. In our case, we exploit this to compile deployment commands from ADL descriptions for configuring all nodes in the JADE management domain. The same ADL descriptions are used to generate meta-components that comprise the system representation. Figure **??** illustrates the initial deployment process (assuming nodes are available): the deployment service is implemented by the configuration manager component, the component factories are generic factories which can instantiate components from ADL descriptions.

6.4 Manager component

Figure 4 illustrates the structure of the management component that was deployed in our experiments. Notice here that this is only an example of a management component that can be easily defined thanks to Jade flexibility.

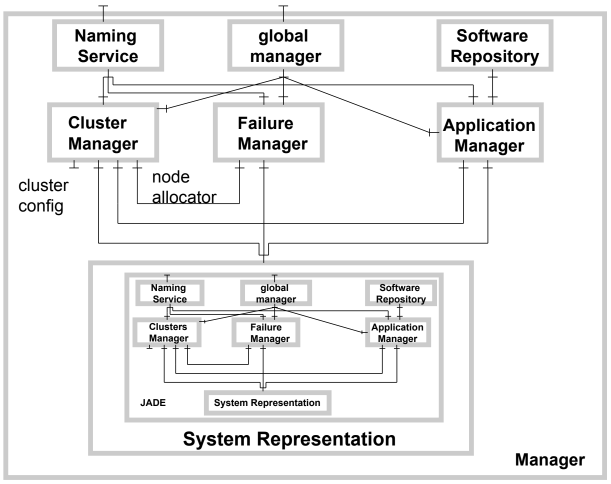

Figure 4. Manager component

The main components of this architecture are Managers, which implement administration functions for particular concerns. Each Manager maintains its own representation of some administrated entities. In our experiment, the addressed concerns are respectively clusters management, failure management and application management The administrated entities, which are represented by components, can be managed by any of the previous manager from their respective points of view. For instance, an application component (e.g. Tomcat) can be managed by the Cluster Manager which is responsible for its location in the administrated cluster, and also managed by the Application Manager which is responsible for it use in the J2EE architecture.

Cluster Manager . The Cluster Manager implements a policy regarding the allocation of nodes and the placement of the administrated entities on these nodes. Each node is represented by a component, which includes each entity deployed on this node. For instance, a probe instantiated by the failure manager or a middleware server (e.g. Tomcat) managed by the application manager, will be shared by a node component if deployed on that node.Arbitrary cluster management policies can be defined, including management of sub-clusters and the management of virtual cluster or nodes.

Application Manager . The Application Manager manages the architecture of the application. In our J2EE scenario, it provides functions for deploying an initial application architecture which includes Apache, Tomcat and MySql components and their bindings. It also provides basic reconfiguration operations which may be used by other managers, e.g. for adding or replacing one component for scalability or repair. Each of these application components is inserted in a node components (managed by the Cluster Manager) when deployed on a particular node.

Failure Manager . The failure Manager implements a policy regarding repair management. It dynamically deploys and configures control loops which implements repair facilities. Control loops are composed of probe, repair manager and actuator components. In case of failure, a repair manager component invokes interfaces of the Application Manager in order to reconfigure the application.

System Representation . As previously described, the System Representation provides a fault tolerant view of the overall system architecture. In case of failure, it is used by a repair manager component to discover the state of the administrated system (including Jade itself) prior to failure.

6.5 JADE infrastructure

JADE is built on a software infrastructure which comprises: the transport system mentioned in section 4.2 that allows the communication of notifications and management commands, the atomic broadcast protocol mentioned in section 5, the node manager component (which in turns provides basic services to identify, allocate and bootstrap machines in a JADE cluster). The transport system and the atomic broadcast protocol are built using the DREAM library [23], a FRACTAL-based communication library, which allows the construction of dynamically configurable, resource-aware, communication subsystems. In principle, this allows the JADE infrastructure itself to be managed by JADE.

6.6 Repair Management

Our current implementation of repair management in JADE relies on the introspection facilities of FRACTAL in order to observe the architecture of the system. When a failure is notified to a repair manager component by a probe, the repair algorithm uses FRACTAL introspection interfaces to access the software architecture of the system (in the System Representation) and find which components were affected by the failure. A node is allocated and the affected components are re-deployed on the allocated machine, and these components are integrated in the overall software architecture, thus reaching a global architecture equivalent to the one preceding the failure. Notice here that repairing this overall system architecture implicitly updates each manager's state consistently. This algorithm is not tied to a particular application environment, since it applies to any distributed Fractal component architecture. Furthermore, JADE being structure in terms of FRACTAL components, the repair algorithm applies to JADE itself in the sense that it can repair JADE managers whenever affected by a failure.

6.7. Example scenario

Figure 3 presents a simple traditional J2EE configuration, involving Apache, Tomcat, Jonas, and MySQL. Consider a scenario where the machine hosting the Tomcat server fails. The failure of the Tomcat server is notified by a sensor node failure detector to the repair manager. The repair manager checks, in the System Representation, the configuration of the failed node prior to failure (in this case a Tomcat server component and its associated bindings). The repair manager requests a new machine from the node manager, deploys a tomcat server on the new machine, and configures it (including setting up its bindings to the Apache Web server and the Jonas Server). Finally, the repair manager sets up a node failure detector sensor for the new machine, and updates the System Representation accordingly.

7 Evaluation

7.1 Qualitative assessment

This section assesses the adequacy of JADE regarding the requirements enounced in section 2[7].In its current version, JADE only deals with **silent hardware failures**. However, since it relies on and benefits from the FRACTAL

[7]The main requirements enumerated in this section 2 are quoted in bold font in the text below

component model, it can easily be extended to deal with **network or software failures**. For instance, a software failure detector can be implemented by a controller monitoring the behaviour of the component it controls.The implemented scenario demonstrates a **basic repair policy** where a tier affected by a machine crash can be dynamically redeployed to restore the application in its initial state.Moreover, we have shown that by duplicating the repair management system and by using an adequate group protocol, it is possible to build a scalable **self-healing repair management system**. We have not yet implemented the duplication of the management system, but we already have an implementation of the required group communication library.**Dynamic system configuration** is systematic in JADE. An application managed by JADE is wrapped by FRACTAL components and the repair system is developed with Java-based FRACTAL components. Therefore, both the application and the system benefit from FRACTAL dynamic configuration.FRACTAL also provides an *explorer* that can be used as a management console providing **support for ad-hoc administration**.We have shown that FRACTAL allows wrapping **legacy software** by implementing a set of controllers. In particular, we have detailed how controllers can be developed to wrap the tiers found in clustered J2EE applications. These controllers provide a programmatic interface on the legacy software, exploited by JADE for dynamic configuration and repair management.JADE being designed in terms of FRACTAL components, any of its constituents can be replaced (at any level of abstraction) to implement a specific policy. In particular, the deployment and repair management services can be adapted to specific needs. In this way, JADE enforces **separation between mechanisms and policies**.Finally, it is important to note that the JADE design strives to **minimize interference with application performance**. Indeed, bindings between wrapped components are reified (using a binding controller), but, as shown in our J2EE experiment, none of the communication between components are intercepted.Overall, this assessment of JADE shows that its design (and more precisely its systematic reliance on the FRACTAL component model) keeps dynamicity and flexibility at the highest level.

7.2 Quantitative Evaluation

We have conducted a series of experiments in order to determine the costs and benefits of using the JADE repair management architecture in a close-to-reality J2EE scenario. The results prove that JADE can handle consecutive system failures and automatically restore system performance without inducing a significant runtime overhead.

7.2.1 Environment

The tests have been carried out using 7 dedicated machines, connected via 1 Gigabit Ethernet. All machines were dual Itanium 900 Mhz servers with 3 GB of RAM and 10000 rpm SCSI drives, running Linux kernel 2.4 SMP ia64. The servlet-only version of RUBiS [24] was used for performance evaluation. RUBiS is an auction application prototype, modeled after eBay with the purpose of serving as a performance benchmark for application servers [11]. The J2EE architecture employed a web tier, a middleware tier and a database tier. The web tier had one Apache machine for serving the static HTML pages. The middleware tier consisted of four Tomcat machines for executing the servlets and the database tier had one MySQL machine for hosting the RUBiS database. The Apache machine was set up to use round-robin load balancing for servlet requests sent to the Tomcat servers. All Tomcat machines were configured to use the Sun JVM version 1.4.2_07-b05 for ia64 architectures. The JVM startup parameters were `"-server -Xmx512m -Xint"` that had the effects of specifying 512 MB for the Java memory allocation pool and of disabling the JVM HotSpot adaptive compiler optimizations. The decision to disable the HotSpot optimizations was taken in order to emphasize the performance evolution of the servers during the tests while using a limited number of machines. Had the compiler optimizations not been disabled, we would have needed a larger amount of database servers and client machines in order for our architecture to support a higher load since the optimized Tomcat machines would have outperformed a single database server. We believe that this choice does not impact on the validity of our experiments as, in any appropriately-sized commercial deployment with enough database servers, when client load increases, the CPU usage of the middleware servers

	Original	JADE
Throughput [req/sec]	82.75	82.75
Avg. Response Time [ms]	194.5	214.75

Table 1. Runtime Overhead

will increase as well, regardless of HotSpot optimizations. This choice merely allowed us to observe the same effects with a smaller hardware deployment. All tests used the RUBiS client emulator running on a dedicated client machine to stress-test the servers. The load consisted of 600 simultaneous clients performing a selection of operations including browsing a product database and placing auction bids on selected items [11]. Each test was timed to include a 500s ramp-up time, 4000s duration and 500s cool-down period. The CPU measurements were collected by the RUBiS client emulator using the widely available *sar* tool part of the *sysstat* [20] suite that is found in most Linux distributions. The values for throughput and average response times were computed by additional logic added to the client emulator. They refer to HTTP requests sent by the virtual clients to the web-tier. Such requests include static HTML elements and servlets and they were generated during the tests using the default RUBiS transition table [24] [11].

7.2.2 JADE **Management Overhead**

In order to determine the overhead imposed by JADE, we have first run the RUBiS benchmark in an unmanaged environment (i.e. without JADE) and then in a JADE managed environment. The comparison of average response time and throughput values obtained in the two cases is presented in 1.

The results show that using JADE induces a small overhead which is due to the periodic probing of the registered components that correspond to the managed servers and the processing of received data. The next section shows that this overhead remains relatively constant even in server-crash conditions.

7.2.3 Automatic Failure Recovery

In order to evaluate the automatic repair functionality of JADE, server crashes were induced in the deployed system. Three Tomcat servers were crashed consecutively, each after a quarter of the test duration (excluding the warm-up time). This eventually resulted in only one server without a crash. All tests were repeated three times and results averaged out. Figure 5 shows the evolution of CPU usage for the Tomcat servers in the described scenario, when the application runs without being managed by JADE. It can be seen that after each server crash, the remaining servers compensate accordingly and their CPU usage increases to accommodate the constant client load. After the third crash, the single remaining server is almost saturated and this reflects significantly in the response times experienced by the users. In extreme conditions, the consecutive or simultaneous crash of a number of servers can lead to complete saturation and loss of availability of the remaining machines, with the effect that client requests are rejected at the web tier level.

In contrast to the above situation, when JADE manages the running environment, the CPU load remains constant with the exception of isolated usage peaks corresponding to server redeployment (see figure 6). Note that for the purpose of the tests, the repair manager used the same machines to restore the crashed Tomcat server. This however is not mandated by the JADE functionality and other available machines in the cluster could be automatically activated as replacements. Since the crashed servers are immediately identified by the repair manager, they can be quickly restored and the remaining healthy machines do not need to compensate and increase their CPU load.

The table 2 shows the evolution of average response time and throughput for each of the test segments (corresponding to the periods before crashes of Tomcat servers). As expected these performance values degrade with each server crash in the case of the unmanaged system, with a significant degradation corresponding to the situation in which only one server remains active. However, when using JADE, the performance characteristics remain

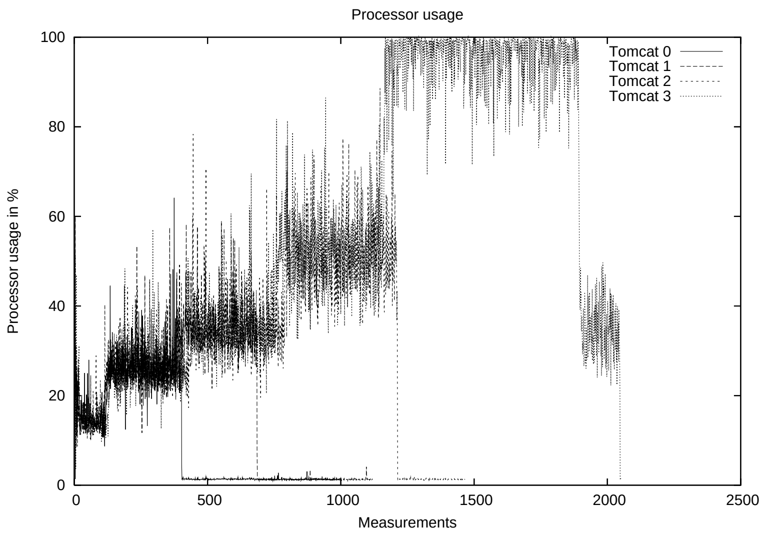

Figure 5. CPU Evolution Without JADE

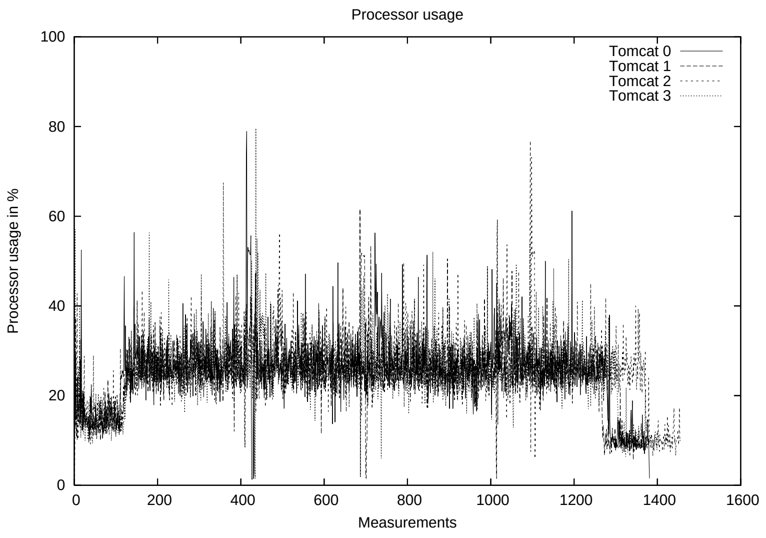

Figure 6. CPU Evolution With JADE

	Throughput Orig.	Throughput JADE	Resp. Time Orig.	Resp. Time JADE
4 servers	82	83	213	239
3 servers	83	83	241	303
2 servers	83	82	266	304
1 servers	77	83	879	310

Table 2. Performance Parameters - Server Crashes

approximately constant throughout the entire duration of the test. The only changes in performance are due to the short repair operations which require the redeployment of the appropriate software environment. We envisage that for long running systems where crashes are not as frequent as in our scenario, the overhead induced by the repair operations will account for even less overall impact. The automatic, short and contained repair operations, together with the low overhead induced when the system is healthy (see 7.2.2) suggest that JADE could contribute to achieving the goal of automatic management of long running, unattended enterprise systems.

8 Related work

Our work is related to several different research domains. We single out the following ones: architecture-based self-healing systems, distributed configuration management, J2EE application server management and cluster and large scale distributed systems management.

Architecture-based self healing systems. Our repair management architecture is an example of architecture-based management [14], in that management functions rely on, and exploit an explicit, causally connected system model which is essentially an architectural description of the running system. There are several works that have adopted an architecture-based approach to system management in the recent years, e.g. [25, 19, 5]. Among those, the work which is closest to ours is the Darwin-based system described in [19]. There are two main differences between this work and ours. First, we exploit the FRACTAL component model [7], which provides more flexibility and run-time adaptability than Darwin. Second, our repair management system exhibits a reflexive structure, where key management infrastructure services (including monitoring, transport, atomic broadcast, and distributed configuration) are themselves component-based, and can be supervised along the same lines as the managed system. In the Darwin-based system, this is not the case. Finally, like JADE the Darwin-based system maintains an explicit system representation (which does not cover infrastructure elements but merely application-level components). JADE, however, does not maintain the same global system representation at each node; instead, JADE maintains a separate view of the system configuration, which reduces the interference between managed system and management system, and alleviates the scaling problems from which the Darwin-based system suffers.

Cluster and large scale distributed systems management. There are numerous works dealing with cluster-based and large scale distributed system management. A large number of them, e.g. [3, 12, 2, 18], deal mostly with resource management, and focus less on recovery management. A work which is closer to ours is the work on the BioOpera system [4], which features an explicit representation of the managed system (called "cluster-awareness", in the form of instrumented workflows, representing application tasks, and node status information), and a recovery management system based on task checkpointing. There are however several differences between our work and BioOpera. First, one can note that the system representation in BioOpera is only derived systematically for application workflows. The cluter system instrumentation is ad-hoc, and thus does not provide for the general framework for repair that we have presented. Second, in contrast to our repair management system, where arbitrary configuration changes can be taken into account, dynamic reconfiguration in BioOpera appears limited to the introduction of new workflows and the handling of node or workflow failures. As a final remark, we can note that the task monitoring and task recovery service which BioOpera provides can be readily provided in our system, provided that the managed components provide a refined life-cycle controller.

Distributed configuration management. A lot of work has also been conducted on distributed configuration management [30, 6, 26, 28]. The closest work to ours [22] presents a model for reifying dependencies in distributed component systems. They consider two kinds of dependencies: *prerequisites* that correspond to the requirements for deploying a component, and *dynamic dependencies* between deployed components. Each component is associated with a *component configurator* which is responsible for ensuring that its dependencies are satisfied both at deployment and reconfiguration time. JADE differs by three main characteristics:

- The system representation used in JADE is *more expressive*: beside component dependencies which correspond to binding and containment relationships in FRACTAL, the system representation allows describing component attributes and could easily be extended to support, for instance, the description of components' behaviour.
- JADE being built using FRACTAL, it benefits from its hierarchical component model, which allows uniform description of the managed system (from low level resources to applicative components). Thus, systems can be managed at various granularities.
- JADE allows dynamic reconfiguration to be applied to itself, which is not the case for *component configurators* that cannot be reconfigured at runtime.

J2EE management. We can also relate our work to research dealing with availability management in J2EE clusters. A number of works in this area rely on tiers replication, which is both a means to scale up performance and to guarantee high availability. In this vein, experiments were reported for database-tier replication (e.g. C-JDBC [10]), EJB-tier replication (e.g. as in JOnAS [27] and JBoss [17]), servlet-tier replication (e.g. as in mod_jk [29]).Most proposals are statically configured. As a consequence, they do not allow automatic adaptation of the application (e.g. repairs or installations on dynamically allocated machines). Recently, the JAGR system [8] has demonstrated how to handle transient software faults in J2EE components. The motivation of this approach is that in the presence of transient software faults, determining the faulty components of the application and rebooting them may be the unique alternative to rebuild a consistent state of the system. The JAGR approach is complementary to ours (especially in its software fault detection aspects), which provides a general repair management for J2EE servers. Moreover, JAGR does not provide a self-healing management system, relying as it does on a unique reboot server.

9 Conclusion

We have presented in the paper a repair management system, which emphasizes a principled, architecture-based approach to the construction of self-managed systems. We have implemented this system in the JADE prototype, which targets repair management in J2EE application servers. Our work demonstrates how to build self-configuring and self-healing systems, through a combination of reflective component-based design, an explicit control loop design pattern, and management system replication. It should be noted that we exploit reflection at two levels: at the level of individual components, which provide access to their internal configuration through FRACTAL controllers, and at a system-wide level through a causally connected system representation component. We believe this reflective design, and the potential for automation which it demonstrates, to be an original contribution of this paper, which we think is applicable to other system management functions.

Much work of course remains to be done to fully validate this design. First, we need to complement our quantitative evaluation with that of the self-healing structure of JADE. Second, we need to extend our prototype to handle software failures in full by introducing appropriate software failure detection facilities. We plan to exploit the failure detection facilities of the PinPoint system [8] to that effect. Third, we need to extend our current scheme to take into account more complex component life-cycles, allowing distributed activities to be repaired and recovered along with individual components. Finally, we need to assess the scalability of our architecture to larger scale distributed systems. There are several interesting possibilities to consider. One would be to consider hierarchies of management domains organised as we described in the paper, with system representations varying in abstraction level according to their level in the domain hierarchy. Another one, would be to consider super-peer based network overlays for ensuring replication among management systems in the domain hierarchy.

References

[1] T. Abdellatif, E. Cecchet, and R. Lachaize. Evaluation of a Group Communication Middleware for Clustered J2EE Application Servers. In *Proceedings Int. Symposium on Distributed Objects and Applications (DOA 2004)*, 2004.

[2] K. Appleby, S. Fakhouri, L. Fong, G. Goldszmidt, M. Kalantar, S. Krishnakumar, D. Pazel, J. Pershing, and B. Rochwerger. Oceano - SLA based management of a computing utility. In *Proceedings of the 7th IFIP/IEEE International Symposium on Integrated Network Management*, May 2001.

[3] Mohit Aron, Peter Druschel, and Willy Zwaenepoel. Cluster reserves: a mechanism for resource management in cluster-based network servers. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 90–101, Santa Clara, California, June 2000.

[4] W. Bausch, C. Pautasso, R Schaeppi, and G. Alonso. BioOpera: Cluster-Aware Computing. In *Proc. IEEE International Conference on Cluster Computing (CLUSTER 2002)*. IEEE Computer Society, 2002.

[5] G. Blair, G. Coulson, L. Blair, H. Duran-Limon, P. Grace, R. Moreira, and N. Parlavantzas. Reflection, self-awareness and sef-healing in OpenORB. In *Proceedings of the 1st Workshop on Self-Healing Systems, WOSS 2002*. ACM, 2002.

[6] M. Blay-Fornarino, A.-M. Pinna-Dery, and M. Riveill. Towards dynamic configuration of distributed applications. In *Proceedings of the International Workshop on Aspect Oriented Programming for Distributed Computing Systems in association with ICDCS '02*, 2002.

[7] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.B. Stefani. An Open Component Model and its Support in Java. In *Proceedings CBSE '04, LNCS 3054*. Springer, 2004.

[8] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. A Microrebootable System - Design, Implementation, and Evaluation. In *Proceedings OSDI '04*, 2004.

[9] G. Candea, E. Kiciman, S. Zhang, P. Keyani, and A. Fox". JAGR: An Autonomous Self-Recovering Application Server. In *Proc. 5th International Workshop on Active Middleware Services*, 2003.

[10] E. Cecchet, J. Marguerite, and W. Zwaenepoel. C-JDBC: Flexible Database Clustering Middleware. In *FREENIX Technical Sessions, USENIX Annual Technical Conference*, 2004.

[11] Emmanuel Cecchet, Julie Marguerite, and Willy Zwaenepoel. Performance and scalability of ejb applications. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 246–261. ACM Press, 2002.

[12] Jeffrey S. Chase, David E. Irwin, Laura E. Grit, Justin D. Moore, and Sara E. Sprenkle. Dynamic virtual clusters in a grid site manager. In *Proceedings 12th IEEE International Symposium on High Performance Distributed Computing (HPDC'03)*, Seattle, Washington, June 2003.

[13] M. Clarke, G. Blair, G. Coulson, and N. Parlavantzas. An Efficient Component Model for the Construction of Adaptive Middleware. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms (Middleware'01)*, 2001.

[14] E. M. Dashofy, A. van der Hoek, and R. N. Taylor. Towards architecture-based self-healing systems. In *Proceedings WOSS '02*. ACM, 2002.

[15] X. Defago, A. Schiper, and P. Urban. Total Order Broadcast and Multicast Algorithms: Taxonomy and Survey. Technical report, EPFL, 2004.

[16] A. Diaconescu, A. Mos, and J. Murphy. Automatic Performance Management in Component-Based Software. In *Proceedings IEEE Int. Conference on Autonomic Computing (ICAC 2004)*, 2004.

[17] M. Fleury and F. Reverbel. The JBoss Extensble Server. In *Middleware 2003, ACM/IFIP/USENIX International Middleware Conference*, volume 2672 of *Lecture Notes in Computer Science*. Springer, 2003.

[18] Yun Fu, Jeffrey Chase, Brent Chun, Stephen Schwab, and Amin Vahdat. SHARP: an architecture for secure resource peering. In *Proceedings of the nineteenth ACM Symposium on Operating Systems Principles (SOSP'03)*, pages 133–148. ACM Press, 2003.

[19] I. Georgiadis, J. Magee, and J. Kramer. Self-organizing software architecture for distributed systems. In *Proceedings of the 1st Workshop on Self-Healing Systems, WOSS 2002*. ACM, 2002.

[20] S. Godard. Sysstat utilities.

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

[22] F. Kon and R. H. Campbell. *Dependence Management in Component-Based Distributed Systems*. 2000.

[23] M. Leclercq, V. Quema, and Jean-Bernard Stefani. DREAM: a Component Framework for the Construction of Resource-Aware, Reconfigurable MOMs. In *Proceedings of the 3rd Workshop on Reflective and Adaptive Middleware (RM'2004)*, 2004.

[24] ObjectWeb. Rubis: Rice university bidding system.

[25] P. Oriezy, M. Gorlick, R. Taylor, G. Johnson, N. Medvidovic, A. Quilici, D. Rosenblum, and A. Wolf. An Architecture-Based Approach to Self-Adaptive Software. *IEEE Intelligent Systems 14(3)*, 1999.

[26] F. Plasil, D. Balek, and R. Janecek. Sofa/dcup: Architecture for component trading and dynamic updating. In *Proceedings of the International Conference on Configurable Distributed Systems*, 1998.

[27] JOnAS Project. Java Open Application Server (JOnAS): A J2EE Platform. http://jonas.objectweb.org/current/doc/JOnASWP.html.

[28] C. Salzmann. Invariants of component reconfiguration. In *Proceedings of the 7th International Workshop on Component-Oriented Programming (WCOP'02)*, 2002.

[29] G. Shachor. Tomcat documentation. The Apache Jakarta Project. http://jakarta.apache.org/tomcat/tomcat-3.3-doc/.

[30] S. Shrivastava and S. Wheater. Architectural support for dynamic reconfiguration of large scale distributed applications. In *Proceedings of the International Conference on Configurable Distributed Systems*, 1998.

[31] S. White, J. Hanon, I. Whalley, D. Chess, and J. Kephart. An Architectural Approach to Autonomic Computing. In *Proceedings ICAC '04*, 2004.