Autonomic Management of Clustered Applications

Sara Bouchenak
Université Joseph Fourier, Grenoble, France
Sara.Bouchenak@inria.fr

Noël De Palma
Institut National Polytechnique de Grenoble, France
Noel.Depalma@inria.fr

Daniel Hagimont
Institut National Polytechnique de Toulouse, France
Daniel.Hagimont@enseeiht.fr

Christophe Taton
Institut National Polytechnique de Grenoble, France
Christophe.Taton@inria.fr

Abstract

Distributed software environments are increasingly complex and difficult to manage, as they integrate various legacy software with proprietary management interfaces. Moreover, the fact that management tasks are performed by humans leads to many configuration errors and low reactivity.

This paper presents Jade, a middleware for self-management of distributed software environments. The main principle is to wrap legacy software pieces in components in order to provide a uniform management interface, thus allowing the implementation of management applications. Management applications are used to deploy distributed applications and to autonomously reconfigure them as required.

We report our experiments in using Jade for the management of a clustered J2EE application.

KEY WORDS: Autonomic management, Legacy systems, Self-optimization, Cluster, J2EE

1 Introduction

Today's computing environments are becoming increasingly sophisticated. They involve numerous complex software that cooperate in potentially large scale distributed environments. These software are developed with very heterogeneous programming models and their configuration facilities are generally proprietary. Therefore, the management [1] of these software (installation, configuration, tuning, repair ...) is a much complex task which consumes a lot of resources:

- human resources as administrators have to react to events (such as failures) and have to reconfigure (repair) complex applications,
- hardware resources which are often reserved (and overbooked) to anticipate load peaks or failures.

A very promising approach to the above issue is to implement administration as an autonomic software. Such a software can be used to deploy and configure applications in a distributed environment. It can also monitor the environment and react to events such as failures or overloads and reconfigure applications accordingly and autonomously. The main advantages of this approach are:

- Providing a high-level support for deploying and configuring applications reduces errors and administrator's efforts.
- Autonomic management allows the required reconfigurations to be performed without human intervention, thus saving administrator's time.
- Autonomic management is a means to save hardware resources as resources can be allocated only when required (dynamically upon failure or load peak) instead of pre-allocated.

This paper presents Jade, an environment for developing autonomic management software. Jade mainly relies on the following features:

- A component model. Jade models the managed environment as a component-based software architecture which provides means to configure and reconfigure the environment.

[1] we also use the term administration to refer to management operations

- Control loops which link probes to reconfiguration services and implement autonomic behaviors.

We used Jade to implement self-optimization in a clustered J2EE application. Here, self-optimization consists in dynamically increasing or decreasing the number of replicas (at any tier of the J2EE architecture) in order to accommodate load peaks.

The rest of the paper is organized as follows. Section 2 details the context of this work. Section 3 presents the design principles underlying Jade. Section 4 describes the implementation of self-optimization. Results of our experimental evaluation are described in Section 5, while related work is discussed in Section 6. Finally, Section 7 draws our conclusions.

2 Experimental Context: Multi-Tier Internet Services

As experimental environment, we made use of the Java 2 Platform, Enterprise Edition (J2EE), which defines a model for developing web applications [20] in a multi-tiered architecture. Such applications usually receive requests from web clients, that flow through a web server (provider of static content), then to an application server to execute the business logic of the application and generate web pages on-the-fly, and finally to a database that persistently stores data (see Figure 1).

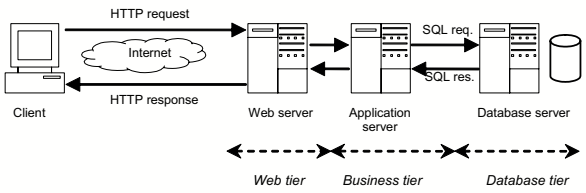

Figure 1. Architecture of dynamic web applications

Upon an HTTP client request, either the request targets a static web document, in which case the web server directly returns that document to the client; or the request refers to a dynamic document, in which case the web server forwards that request to the application server. When the application server receives a request, it runs one or more software components (e.g. Servlets, EJBs) that query a database through a JDBC driver (Java DataBase Connection driver) [21]. Finally, the resulting information is used to generate a web document on-the-fly that is returned to the web client.

In this context, the increasing number of Internet users has led to the need of highly scalable and highly available services. Moreover, several studies show that the complexity of multi-tier architectures with their dynamically generated documents represent a large portion of web requests, and that the rate at which dynamic documents are delivered is often one or two orders of magnitudes slower than static documents [10, 11]. This places a significant burden on servers [7]. To face high loads and provide higher scalability of Internet services, a commonly used approach is the replication of servers in clusters. Such an approach usually defines a particular (hardware or software) component in front of the cluster of replicated servers, which dynamically balances the load among the replicas. Here, different load balancing algorithms may be used, e.g. Random, Round-Robin, etc. Among the existing J2EE clustering solutions we can cite C-JDBC for a cluster of database servers [8], JBoss clustering for a cluster of JBoss EJB servers [6], mod_jk for a cluster of Tomcat Servlet servers [17], and the L4 switch for a cluster of replicated Apache web servers [19].

Clustered multi-tier J2EE systems represent an interesting experimental environment for our autonomic management environment, since they bring together all the challenges addressed by Jade:

- the management of a variety of legacy systems, since each tier in the J2EE architecture embeds a different piece of software (e.g. a web server, an application server, or a database server),
- very complex administration interfaces and procedures associated with very heterogeneous software,
- the requirement for high reactivity in taking into account events which may compromize the normal behaviour of the managed system, e.g. load peaks or failures.

Therefore, we chose the J2EE platform to illustrate the research contributions of this paper.

3 Design Principles and Architecture

The main motivation for Jade is to allow software to be managed by programs (instead of humans). Since the managed software is very hererogeneous, our key design choice was to rely on a component model to provide a uniform management interface for any managed resource.

Therefore, any software managed with Jade is wrapped in a component which interfaces its administration procedures.

The component model that we use (Fractal [5]) is overviewed in Section 3.1. We then describe in Section 3.2

the wrapping of J2EE tiers in Fractal components. Section 3.3 and Section 3.4 respectively present how applications can be deployed and reconfigured, thanks to this component-based approach.

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 reconfiguration 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. An interface is an access point 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. defined as an assembly of several sub-components, or primitive, i.e. encapsulating an executable program.

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.). 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 access to a component internals, allowing for component introspection and the control of component behaviour.

A controller provides a control interface and implements a control behavior for the component, such as controlling the activities in the components (suspend, resume) or modifying some of its attributes. The Fractal model allows for arbitrary (including user defined) classes of controller. It specifies, however, several useful forms of controllers, which can be combined and extended to yield components with different control features. This includes the following controllers :

- 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: for composite components, 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.

Several implementations of the Fractal model have been issued in different contexts, e.g. an implementation devoted to the configuration of operating systems on a bare hardware (Think) or an implementation on top of the Jave virtual machine (Julia) targeted to the configuration of middleware or applications. The work reported in this paper relies on this later implementation of Fractal.

3.2 Component-based management

In order to allow software to be managed by programs (instead of humans), any software managed with Jade is wrapped in a Fractal component which interfaces its administration procedures, through the provision of controllers. Thanks to Fractal's hierachical model, arbitrary sophisticated organizations can be modelized.

This provides a means to:

- Manage legacy entities using a uniform model (the Fractal control interface), instead of relying on software-specific, hand-managed, configuration files.
- Manage complex environments with different points of view. For instance, using appropriate composite components, it is possible to represent the network topology, the configuration of the J2EE middleware, or the configuration of an application on the J2EE middleware.

- Add a control behavior to the encapsulated legacy entities (e.g. monitoring, interception and reconfiguration).

Therefore, the Fractal component model is used to implement a management layer on top of the legacy layer (composed of the actual managed software). In the management layer, all components provide the same (uniform) management interface for the encapsulated software, and the corresponding implementation (the wrapper) is specific to each software (e.g. in the case of J2EE, Apache web server, Tomcat Servlet server, MySQL database server, etc.). The interface allows managing the element's attributes, bindings and lifecycle.

Relying on this management layer, sophisticated administration programs can be implemented, without having to deal with complex, proprietary configuration interfaces, which are hidden in the wrappers. The management layer provides all the facilities required to implement such administration programs:

Introspection. The framework provides an introspection interface that allows observing managed components. For instance, an administration program can inspect an Apache web server component (encapsulating the Apache server) to discover that this server runs on node1:port 80 and is bound to a Tomcat Servlet server running on node2:port 66. It can also inspect the overall J2EE infrastructure, considered as a single composite component, to discover that it is composed of two Apache servers interconnected with two Tomcat servers connected to the same MySQL database server.

Reconfiguration. The framework provides a reconfiguration interface that allows control over the component architecture. In particular, this control interface allows changing component attributes or bindings between components. These configuration changes are reflected onto the legacy layer. For instance, an administration program can add or remove an Apache replica in the J2EE infrastructure to adapt to workload variations.

The above approach is illustrated in Figure 2 in the case of a J2EE architecture. In this setting, an L4-switch balances the requests between two Apache server replicas. The Apache servers are connected to two Tomcat server replicas. The Tomcat servers are both connected to the same MySQL server. The vertical dashed arrows (between the management and legacy layers) represent management relationships between components and the wrapped software entities. In the legacy layer, the dashed lines represent relationships (or bindings) between legacy entities, whose implementations are proprietary. These bindings are represented in the management layer by (Fractal) component bindings (full lines in the figure).

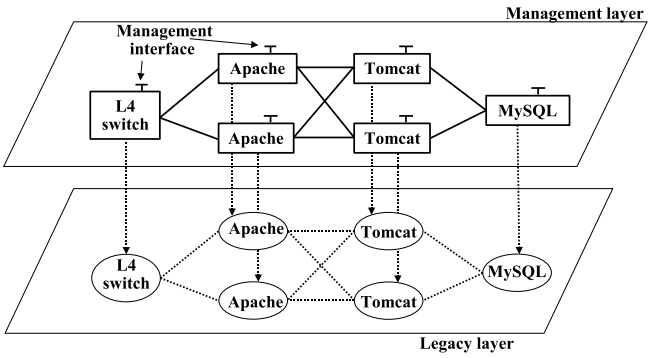

Figure 2. Management layer for a clustered J2EE application

We now give an example of a Fractal wrapper for the Apache server that is part of the J2EE architecture. The wrapper provides an attribute controller, a binding controller and a lifecycle controller:

The attribute controller interface is used to set attributes related to the local execution of the Apache server. For instance, a modification of the port attribute of the Apache component is reflected in the *httpd.conf* file in which the port attribute is defined.

The binding controller interface is used to connect Apache with other middleware tiers. For instance, invoking the bind operation on the Apache component sets up a binding between one instance of Apache and one instance of Tomcat. The implementation of this bind method is reflected at the legacy layer in the *worker.properties* file used to configure the connections between Apache and Tomcat servers.

The life cycle controller interface is used to start or to stop the server as well as to read its state (i.e. running or stopped). It is implemented by calling the Apache commands for starting/stopping a server.

Other servers (Tomcat and MySQL) are wrapped in a similar way into Fractal components, and provide the same management interface. These elements are then used by the J2EE self-optimization mechanism, as described in section 4.

3.3 Deployment

The architecture of an application is described using an Architecture Description Language (ADL), which is one of the basic features of the Fractal component model. This description is an XML document which details the architectural structure of the application to deploy on the cluster, e.g. which software resources compose the multi-tier

J2EE application, how many replicas are created for each tier, how are the tiers bound together, etc ...

A Software Installation Service component (a component of Jade) allows retrieving the encapsulated software resources involved in the multi-tier J2EE application (e.g., Apache Web server software, MySQL database server software, etc.) and installing them on nodes of the cluster. A Cluster Manager component is responsible for the allocation of nodes (from a pool of available nodes) which will host the replicated servers of each tier.

The deployment of an application is the interpretation of an ADL description, using the Software Installation Service and the Cluster Manager to deploy application's components on nodes. The autonomic administration software is also described using this ADL and deployed in the same way. However, this description of the administration software is separated from that of the application.

3.4 Implementing autonomic managers

Autonomic computing is achieved through autonomic managers, which implement feedback control loops. These loops regulate and optimize the behavior of the managed system. Figure 3 illustrates control loops in the Jade autonomic management system. It shows two managers that regulate two specific aspects of the platform (self-recovery, detailed in [4], and self-optimization, discussed in this paper). Each autonomic manager in Jade is based on a control loop that includes sensor, actuator and analysis/decision components.

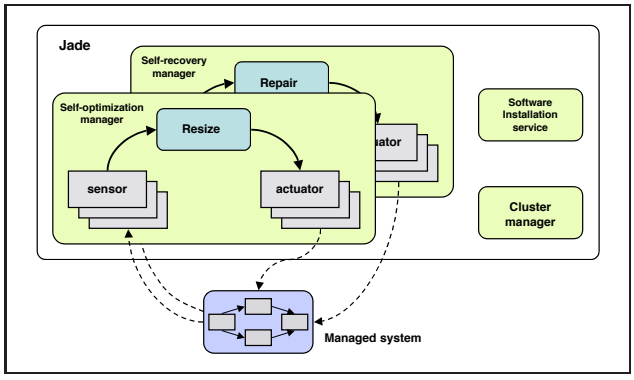

Figure 3. Control loops in Jade

Sensors are responsible for the detection of the occurrence of a particular event, e.g. a QoS requirement violation in case of a self-optimization manager, or an element failure (node, middleware or component) for a self-recovery manager. Sensors must be efficient and lightweight. In the particular case of the self-optimization manager, sensors must monitor and aggregate low-level information such as CPU/memory usage, or higher-level information such as client response times.

Analysis/decision components (or reactors) represent the actual reconfiguration algorithm, e.g. repairing a failed element in case of a self-recovery manager, or resizing the cluster of replicated servers upon load changes in case of a self-optimization manager. Reactors receive notifications from sensors and make use of actuators when a reconfiguration operation is necessary.

Actuators represent the individual mechanisms necessary to implement reconfiguration operations, e.g. allocating a new node to a cluster of replicas, adding/removing a replica to the cluster of replicated servers, updating connections between the tiers.

Sensors, Actuators and Reactors are implemented as Fractal components, which allows reusing and combining them to assemble specific autonomic managers. Moreover, this allows autonomic managers to be deployed and managed using the same Jade framework (Jade administrates itself).

4 Implementation of Self-Optimization

In this section, we discuss the means of implementing self-optimization in replicated cluster-based systems, using the framework described in Section 3. Optimization is defined using two main criteria: performance, as perceived by the clients (e.g. response time) or by the application's provider (e.g. global throughput); and resource usage (e.g. processor occupation). One may then define an "optimal" region using a combination of these criteria. Providing self-optimization for a system consists in maintaining the system in the optimal region, in the presence of dynamic changes, e.g. widely varying workload. Here, we consider implementing self-optimization using resizing techniques, i.e. dynamically increasing or decreasing the number of nodes allocated to the application.

4.1 Self-sizeable clustered application

A standard pattern for implementing scalable clustered servers is the load balancer. In this pattern, a given application server is statically replicated at deployment time and a front-end proxy distributes incoming requests among the replicated servers.

Jade aims at autonomously adjusting the number of replicas used by the application when the load varies. This is implemented by an autonomic manager which implements the required control loops. We implemented two control loops, one devoted to the management of the replicated web container (Tomcat) and the other devoted to the management of the replicated database (MySQL). In both cases

the control loop has the following sensors, actuators and reactors.

The *sensors* periodically measure the chosen performance (or QoS) criteria, i.e. a combination of CPU usage and user-perceived response time. In our expriments with Tomcat and MySQL servers, the used sensor is a probe that collects CPU usage information on all the nodes where such a server is deployed. This probe computes a moving average of the collected data in order to remove artifacts characterizing the CPU consumption. It finally computes an average CPU load across all nodes, so as to observe a general load indication of the whole replicated server.

The *actuators* are used to reconfigure the system. Thanks to the uniform management interface provided by Jade, the actuators are generic, since increasing or decreasing the number of replicas of an application is implemented as adding or removing components in the application structure.

The *reactors* implement an analysis/decision algorithm. They receive notifications from sensors, and react, if needed, by increasing or decreasing the number of replicas allocated to the controlled tier. In our experiments, the decision logic implemented to trigger such a reconfiguration is based on thresholds on CPU loads provided by sensors.

The main operations performed by the reactor when more replicas are required are the following: allocate free nodes for the application, deploy the required software on the new nodes if necessary, perform state reconciliation with other replicas in case of servers with dynamic data, and integrate the new replicas with the load balancer. Similarly, if the clustered server is under-utilized, the main operations performed by the reactor are the following: unbind some replicas from the load balancer, stop these replicas, and release the nodes hosting these replicas if no longer used. To create an additional replica (i.e. node + software server), the reactor uses the services provided by Jade, e.g. the Cluster Manager component to allocate new nodes, the Software Installation Service component to install the required software.

One important issue to address when managing replicated servers with dynamic data (modifiable state) is data consistency. This was not a problem in the case of the web container (Tomcat) as our evaluation application was composed of servlets with no dynamically changing session information. In the case of database servers, the load balancer that we used was C-JDBC; C-JDBC plays the role of load balancer and replication consistency manager [8], each server containing a full copy of the whole database (full mirroring).

To manage a dynamic set of database servers, a newly allocated server must synchronize its state with respect to the whole clustered database before it is activated. To do so, a ”recovery log” has been added to the C-JDBC load-balancer. This recovery log is implemented as a particular database whose purpose is to keep track of all the requests that affect the state of the database. Basically, all write requests are logged and indexed as strings in this recovery log. When a new server is inserted in the clustered database, the state of this server is initially known and is potentially not up-to-date. The recovery log enables us to know the exact set of write requests to replay on this server to make it be up-to-date. Once these requests have been processed by the newly allocated server, we can reinsert it in the clustered database as an active and up-todate replica. Symmetrically, removing a database replica is realized by keeping trace of the state of this replica. This state is stored as the index value in the recovery log corresponding to the last write request that it has executed before being disabled.

4.2 Generality of the approach

One of our objectives when building Jade was to provide an environment for building autonomic managers which manages a variety of specific legacy systems. Jade is built on top of a set of generic components which are assembled in a specific way so as to implement the required autonomic managers.

For instance, the self-optimization autonomic manager is realized by the means of a control-loop which is composed of three kinds of components:

The sensors: the set of sensors that we currently use is rather generic with respect to the J2EE experiment that we conducted. Indeed, they provide estimators mainly related to resource usage and can therefore be used under various contexts.

The decision logic: the logic which is in charge of deciding a reconfiguration policy of the system consists in a set of triggers based on thresholds. Such a logic is generic and can be used in many cases.

The actuators: The wrappers provide a uniform representation of the managed system as a set of components which expose a uniform interface. Thus the actuators are built using this uniform interface and are therefore fully generic.

However, in the context of the J2EE use-case, some components are submitted to some specificities:

Though the wrapper components provide a uniform interface which is used to build the actuators, the implementation of these wrappers is specific to the legacy software they wrap.

The implementation of the decision logic is based on thresholds. This implies a configuration of the various parameters characterizing the thresholds. This configuration is specific to the managed system. For instance, the thresholds of the self-optimization manager have been determined manually with some benchmarks. Note that the determination of these parameters constitutes a key challenge of this

manager.

Even if the set of sensors used in the current prototype is generic, some sensors can be specifically written for a particular aspect we are interested in. For example, a sensor specific to optimization may provide an estimator of the response-time to client requests. However the CPU was known to be the bottleneck resource as far as our J2EE system was concerned.

5 Evaluation

This section describes a qualitative and quantitative evaluation conducted with Jade.

5.1 Qualitative evaluation

In this section, we show the benefits of using Jade to perform system reconfiguration, compared to an ad-hoc approach. Figure 4 illustrates a scenario where, initially, an Apache web server (Apache1) is running on node1, and connected to a Tomcat Servlet server (Tomcat 1) running on node2. In this scenario we want to reconfigure the clustered middleware layer by replacing the connection between Apache1 and Tomcat1 by a connection between Apache1 and a new server Tomcat2.

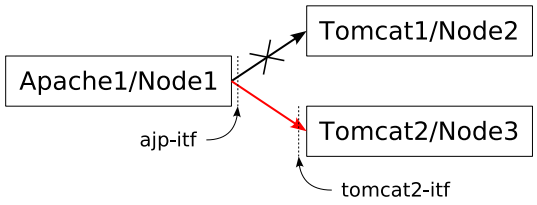

Figure 4. Reconfiguration scenario

Without the Jade infrastructure, this simple reconfiguration scenario requires the following steps to be manually done (with very low reactivity), in a legacy-dependent way: first log on node1, then stop the Apache server by running the Apache *shutdown* script, then edit and update the configuration file (*worker.properties*) in Apache to specify its binding to the new Tomcat server (Tomcat2 on node3) as follows:

```
worker.worker.port=8098
worker.worker.host=node3
worker.worker.type=ajp13
worker.worker.lbfactor=100
worker.list=worker, loadbalancer
worker.loadbalancer.type=lb
worker.loadbalancer.balanced_workers=worker
```

Finally, the Apache server is restarted by running the *httpd* script.

With Jade, as soon as the required wrappers have been implemented, such a reconfiguration can easily be implemented in an administration application. The operations required to perform this same reconfiguration are simply few operations on the involved components in the management layer, namely:

```
Apache1.stop()
  // unbind Apache1 from Tomcat1
Apache1.unbind("ajp-itf")
  // bind Apache1 to Tomcat2
Apache1.bind("ajp-itf",tomcat2-itf)
  // restart Apache1
Apache1.start()
```

5.2 Quantitative evaluation

Testbed application

The evaluation of the self-optimization management implemented in Jade has been realized with RUBiS [1], a J2EE application benchmark based on servlets, which implements an auction site modeled over eBay. It defines 26 web interactions, such as registering new users, browsing, buying or selling items. RUBiS also provides a benchmarking tool that emulates web client behaviors and generates a tunable workload. This benchmarking tool gathers statistics about the generated workload and the web application behavior.

Hardware environment

The experimental evaluation has been performed on a cluster of x86-compatible machines. The experiments required up to 9 machines: one node for the Jade management platform, one node for the load-balancer in front of the replicated web/application servers, up to two nodes for the replicated web and application servers, one node for the database load-balancer, up to three nodes for the replicated database servers, one node for the client emulator (which emulates up to 500 clients). The number of nodes actually used during these experiments varies, according to the dynamic changes of the workload, and thus to the dynamic resizing of the application. All the nodes are connected through a 100Mbps Ethernet LAN to form a cluster.

Software Environment

The nodes run various versions of the Linux kernel. The J2EE application has been deployed using open source middleware solutions: Jakarta Tomcat 3.3.2 [22] for the web and servlet servers, MySQL 4.0.17 [13] for the database servers, C-JDBC 2.0.2 [8] for the database load-balancer, PLB 0.3 [15] for the application server load-balancer. We used RUBiS 1.4.2 as the running J2EE application. These experiments have been realized with Sun's JVM JDK 1.5.0.04. We used the MySQL Connector/J 3.1.10 JDBC driver to connect the database load-balancer to the database servers.

Evaluation Scenario

In order to evaluate the performance optimization aspect of the Jade management platform, we have designed a scenario that illustrates the dynamic allocation and deallocation of nodes to tackle performance issues related to a changing workload. This scenario is described below.

We aim at showing the dynamic allocation and deallocation of nodes in response to workload variations. Therefore we have submitted our managed J2EE system to the following workload: (i) at the beginning of the experiment, the managed system is submitted to a medium workload: 80 emulated clients; then (ii) the load increases progressively up to 500 emulated clients: 21 new emulated clients every minute; finally (iii) the load decreases symmetrically down to the initial load (80 clients).

Initially, the J2EE system is deployed with one application server (Tomcat) and one database server (MySQL). The optimization manager reacts to the load variation by allocating and freeing nodes, as described below.

In this experiment, we have deployed two instances of a control loop in order to tackle performance issues of the clustered database and the clustered application server. Periodically, the resource usage of the nodes participating to the managed service is monitored. In practice, the sensor of the control loops gathers the CPU usage of these nodes every second and computes a spatial (over these nodes) and temporal (over the last period) average CPU usage value. This average CPU usage value is compared to minimum and maximum thresholds. The objective is to keep the CPU usage value between these two thresholds. Therefore, if this value is over the maximum threshold, the replicas are overloaded and the control loop deploys a new replica on a free node. On the contrary, if this value is under the minimum threshold, the replicas are under-used, and the control loop removes one node hosting a replica of the managed service.

The two control loops are executed independently. However, in order to prevent oscillations, a reconfiguration started by one of the control loops inhibits any new reconfiguration for a short period (one minute).

The control loop execution is realized every second. This time interval is short to quickly detect performance variations and to react promptly to them. In order to have a consistent load indicator, the CPU usage is smoothed by a temporal average (moving average). The strength of this average is experimentally fixed accordingly to the variability of the CPU usage observed during benchmarking experiments. For instance, the average CPU usage is computed over the last 60 seconds for the application servers and over the last 90 seconds for the database servers.

Finally the thresholds used to trigger the reconfigurations have also been determined experimentally through specific benchmarks. They have been adjusted so that the reconfigurations are triggered at appropriate moments. For instance, the maximum thresholds have been determined so that the response time for clients' requests remains acceptable when the reconfigurations start.

Autonomic Reconfiguration

Figure 5 shows the effect of the control loop on the number of replicas, for both the application and database servers. This behavior may be explained as follows. As the workload progressively increases (180 clients), the average resource consumption of the clustered database also increases, which triggers the allocation of one new database server. The system now contains two database servers. The workload continues to grow (320 clients), and triggers a second node allocation for the clustered database. Thus the system is here composed of one Tomcat server and three MySQL database servers. The workload increases further (420 clients). Now the resource consumption of the Tomcat servers reaches a threshold, which triggers the allocation of one new Tomcat server. The system is now composed of two Tomcat servers and three MySQL databases. The workload then increases (500 clients) without saturating this configuration, and then starts decreasing. The workload decrease (400 clients) implies a decrease of the resource consumption of the Tomcat servers, which triggers the deallocation of one Tomcat server. The workload decrease continues (280 clients) and implies this time a low resource consumption of the clustered database, which triggers the deallocation of one database server.

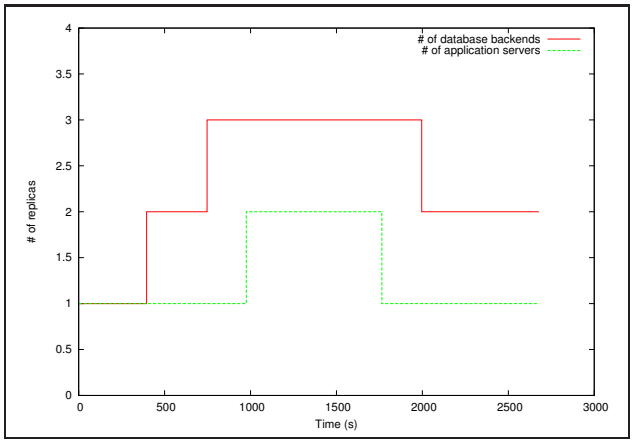

Figure 5. Dynamically adjusted number of replicas

CPU Usage

To quantify the effect of the reconfigurations, this scenario (the workload) has also been experimented without Jade, i.e. without any reconfiguration, so that the managed system is not resized. Figure 6 and Figure 7 present the results of these experiments and show the thresholds used to

trigger dynamic reconfigurations.

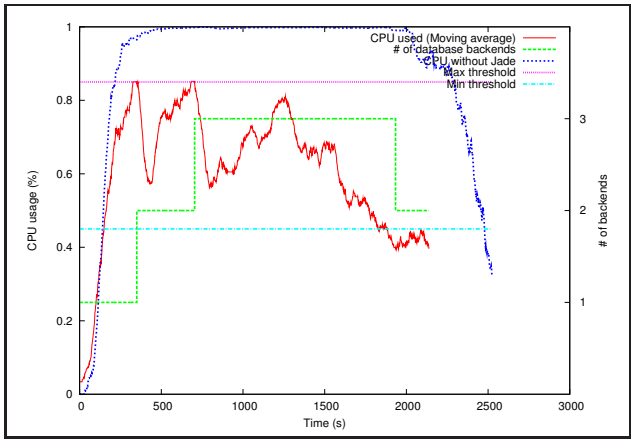

Figure 6. Behavior of the database tier

One of the control loops is dedicated to the performance optimization of the clustered database. When the average CPU usage reaches the maximum threshold set for the database, the control loop triggers the deployment of a new database server, which implies a decrease of the average CPU usage. Symmetrically, when the average CPU usage gets under the minimum threshold, the control loop triggers the removal of one server. This behavior is quite visible in Figure 6. The contrast with the static case of a system that is not resized is obvious: as the workload increases, the CPU usage eventually saturates. This results in a thrashing of the database, which stops when the load decreases.

The second control loop is dedicated to the performance optimization of the clustered application server. The behavior of the control loop is identical to that described above. However the comparison with a system that is not dynamically resized must be correlated with the thrashing that affects the database. In Figure 7, since the database is already saturated, the application servers spend most of the time waiting for the database. This explains why the CPU usage measured during high loads remains moderate.

Response time

We now consider the impact on performance in terms of client request response times. Figure 8 and Figure 9 show the client response time in Rubis, comparing the results of the evaluation of the J2EE system when it is not managed by Jade with the same system when self-optimized with Jade. In both cases, the workload was increased by a factor of 5, and then decreased by a factor of 5. Here, Jade was able to maintain the web client perceived response time stable (around 590 ms in average), while the same experiment run without Jade results in a continuously increasing client response time (10.42 s in average) when the workload increases.

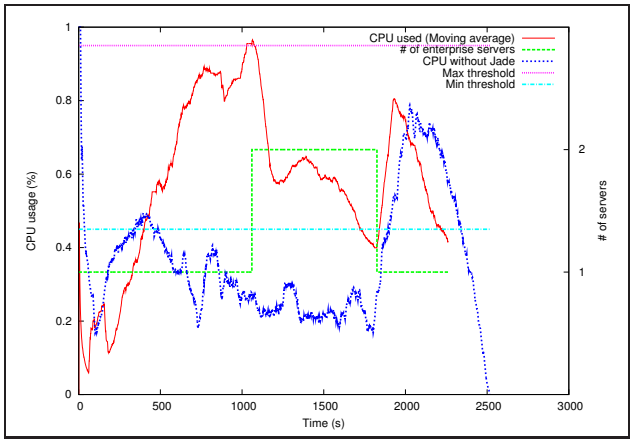

Figure 7. Behavior of the application tier

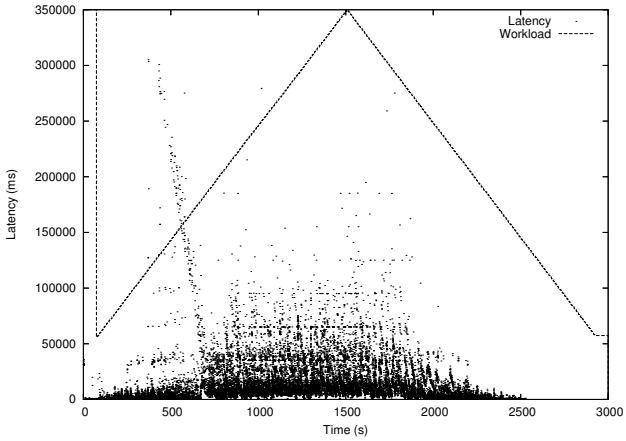

Figure 8. Response time without Jade

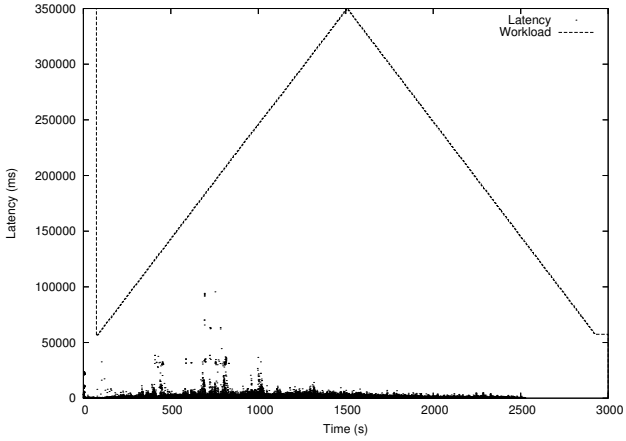

Figure 9. Response time with Jade

Intrusivity

We also evaluated the intrusivity induced by the Jade management system. The intrusivity has been measured by comparing two executions of the J2EE application: when

it is run and managed by Jade and when it is run by hand, without Jade. During this evaluation, the J2EE application has been submitted to a medium workload so that its execution under the control of Jade didn't induce any dynamic reconfiguration. This intrusivity is quantified in terms of average throughput, response time, processor and memory usage of the J2EE application in Table 1.

	with Jade	without Jade
Throughput (req./s)	12	12
Resp.time (ms)	89	87
CPU usage (%)	12.74	12.42
Memory usage (%)	20.1	17.5

Table 1. Performance overhead

The processor and memory usage are computed as average values over all cluster nodes involved in this experiment. These results show no significant overhead as far as performance is concerned. We can notice a slight memory overhead (20.1% vs. 17.5%) that can be linked with the management components which are deployed on every node when Jade is active. However, Jade does not induce a perceptible overhead on CPU usage. This is due to the fact that Jade does not intercept communications at the application level. The wrappers only interface administration procedures.

6 Related Work

Autonomic computing is an appealing approach that aims at simplifying the hard task of system management, thus building self-healing, self-tuning, and self-configuring systems [12].

Management solutions for legacy systems are usually proposed as ad-hoc solutions that are tied to particular legacy system implementations [18, 25]. This unfortunately reduces reusability of management policies and requires these policies to be reimplemented each time a legacy system is taken into account in a particular context. Moreover, the architecture of managed systems is often very complex (e.g. multi-tier architectures), which requires advanced support for its management. Projects such as Jade or Rainbow [9], with a component-based approach, propose a generic way to manage complex system architectures.

Several projects have addressed the issue of self-optimization and resource management in a cluster of machines. Instead of statically allocating resources to applications managed in the cluster (which would lead to a waste of resources), they aim at providing dynamic resource allocation.

In a first category of projects, the software components required by any application are all installed and accessible on any machine in the cluster. Therefore, allocating additional resources to an application can be implemented at the level of the protocol that routes requests to the machines (Neptune [18] and DDSD [26]). Some of them (e.g. Cluster Reserves [3] or Sharc [24]) assume control over the CPU allocation on each machine, in order to provide strong guarantees on resource allocation.

In a second category of projects, the unit of resource allocation is an individual machine (therefore applications are isolated, from a security point of view). A machine may be dynamically allocated to an application by a hosting center, and the software components of that application must be dynamically deployed on the allocated machine. Projects like Jade, Oceano [2], QuID [16], OnCall [14], Cataclysm [23] or [25] fall into this category.

7 Conclusion

Multi-tier platforms are now widely used to build Internet application servers. These platforms are usually replicated on clusters of machines to achieve scalability and availability. Managing such systems is an increasingly complex task, which autonomic computing is believed to alleviate. Many real applications include legacy systems, with limited ability for fine-grained control, which adds to the difficulty of the task.

We have designed and implemented Jade, an environment for autonomic management of legacy systems. The main contribution of this paper is the definition of an architectural framework for constructing flexible and efficient autonomic management facilities for such systems. To prove the validity of our approach, we have applied this framework to the self-optimization of J2EE applications in the face of the wide load variations observed in Internet applications

Jade mainly relies on two features. First, a component model is used to implement a management layer in which the administrated software are wrapped. This management layer provides the administered software with a uniform management interface, allowing autonomic programs to be built on. Second, Jade provides a framework for building autonomic managers, which allows regulating a set of managed software for a specific management aspect.

The uniform management interface of Jade greatly simplifies the development of autonomic managers as it hides the complexity of heterogeneous configuration files. At the component level, adding or removing a servlet server component is done in the same way as adding or removing a database.

As a testbed for Jade, we have implemented a simple instance of a self-optimizing version of an emulated electronic auction site [1] deployed on a clustered J2EE plat-

form. We have used a dynamic capacity provisioning technique in which each replicated tier of the middleware layer is able to dynamically acquire or release servers to react to load variations. Specifically, we showed that dynamic provisioning of nodes, using a very simple threshold-based control algorithm, helps regulating the load on the servers at different tiers, and thus protects the users again performance degradation due to overload, while avoiding static reservation of resources in the cluster. This is only a first step, whose main intent was to demonstrate the ability of Jade to implement specific autonomic components for various aspects.

Part of our future work will focus on improving the self-optimizing algorithm by setting incrementally and dynamically its parameters. Furthermore we intend to work on the problem of conflicting autonomic policies. Managers have their own goal and control loops and therefore require a way to arbitrate potential conflicts. The component-based approach gives us a way to build an infrastructure as a set of hierarchical and interconnected components, which may help implementing policy arbitration managers. We also intend to apply our self-optimization techniques on other use cases to show the genericity of our approach.

References

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

[2] K. Appleby, S. Fakhouri, L. Fong, G. Goldszmidt, and M. Kalantar. Oceano - SLA based management of a computing utility. In *7th IFIP/IEEE International Symposium on Integrated Network Management*, Seattle, WA, May 2001.

[3] M. Aron, P. Druschel, and W. Zwaenepoel. Cluster Reserves: a mechanism for resource management in cluster-based network servers. In *International Conference on Measurement and Modeling of Computer Systems (ACM SIGMETRICS-2000)*, Santa Clara, CA, June 2000.

[4] 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, Oct. 2005.

[5] E. Bruneton, T. Coupaye, and J. B. Stefani. Recursive and Dynamic Software Composition with Sharing. In *International Workshop on Component-Oriented Programming (WCOP-02)*, Malaga, Spain, June 2002. `http://fractal.objectweb.org`.

[6] B. Burke and S. Labourey. Clustering With JBoss 3.0. Oct. 2002. `http://www.onjava.com/pub/a/onjava/2002/07/10/jboss.html`.

[7] E. Cecchet, A. Chanda, S. Elnikety, J. Marguerite, and W. Zwaenepoel. Performance Comparison of Middleware Architectures for Generating Dynamic Web Content. In *4th ACM/IFIP/USENIX International Middleware Conference*, Rio de Janeiro, Brazil, June 2003.

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

[9] D. Garlan, S. Cheng, A. Huang, B. Schmerl, and P. Steenkiste. Rainbow: Architecture-based self adaptation with reusable Infrastructure. *IEEE Computer*, 37(10), 2004.

[10] X. He and O. Yang. Performance Evaluation of Distributed Web Servers under Commercial Workload. In *Embedded Internet Conference 2000*, San Jose, CA, Sept. 2000.

[11] A. Iyengar, E. MarcNair, and T. Nguyen. An Analysis of Web Server Performance. In *IEEE Global Telecommunications Conference (GLOBECOM'97)*, Phoenix, AR, Nov. 1997.

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

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

[14] J. Norris, K. Coleman, A. Fox, and G. Candea. OnCall: Defeating Spikes with a Free-Market Application Cluster. In *1st International Conference on Autonomic Computing (ICAC-2004)*, May 2004.

[15] PLB. PLB - A free high-performance load balancer for Unix. `http://plb.sunsite.dk/`.

[16] S. Ranjan, J. Rolia, H. Fu, and E. Knightly. QoS-Driven Server Migration for Internet Data Centers. In *10th International Workshop on Quality of Service (IWQoS 2002)*, Miami Beach, FL, May 2002.

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

[18] K. Shen, H. Tang, T. Yang, and L. Chu. Integrated resource management for cluster-based internet services. In *5th USENIX Symposium on Operating System Design and Implementation (OSDI-2002)*, Dec. 2002.

[19] S. Sudarshan and R. Piyush. Link Level Load Balancing and Fault Tolerance in NetWare 6. NetWare Cool Solutions Article. Mar. 2002. `http://developer.novell.com/research/appnotes/2002/march/03/a020303.pdf%`.

[20] Sun Microsystems. Java 2 Platform Enterprise Edition (J2EE). `http://java.sun.com/j2ee/`.

[21] Sun Microsystems. Java DataBase Connection (JDBC). `http://java.sun.com/jdbc/`.

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

[23] B. Urgaonkar and P. Shenoy. Cataclysm: Handling Extreme Overloads in Internet Services. Technical report, Department of Computer Science, University of Massachusetts, Nov. 2004.

[24] B. Urgaonkar and P. Shenoy. Sharc: Managing CPU and network bandwidth in shared clusters. *IEEE Transactions on Parallel and Distributed Systems*, 15(1), 2004.

[25] B. Urgaonkar, P. Shenoy, A. Chandra, and P. Goyal. Dynamic Provisiong of Multi-Tier Internet Applications. In *2nd International Conference on Autonomic Computing (ICAC-2005)*, Seattle, WA, June 2005.

[26] H. Zhu, H. Ti, and Y. Yang. Demand-driven service differentiation in cluster-based network servers. In *20th Annual Joint Conference of the IEEE Computer and Communication Societies (INFOCOM-2001)*, Anchorage, AL, Apr. 2001.