Autonomic Management of Internet Services: Experience with Self-Optimization

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

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

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

Sacha Krakowiak
Université Joseph Fourier
Grenoble, France
Sacha.Krakowia@inria.fr

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

Abstract

Autonomic computing is an appealing approach to facilitate the management of distributed systems. However, several challenges remain open among which: (i) How to manage legacy systems? (ii) How to manage complex distributed systems? In this paper, we precisely address these issues. The paper presents the design, implementation and evaluation of Jade, an environment for autonomic management of legacy computing systems. This environment relies on two main frameworks: a framework for administrable elements, which provides the administered legacy system's elements with a uniform management interface; and a framework for autonomic managers, which regulates a set of managed elements for a specific management aspect. The aspect that we consider in this paper is self-optimization, i.e. preserving the system's performance in response to wide variations of the load.

The main contribution of this paper is a generic architectural model that eases the construction of autonomically managed systems. Thus, integrating a new managed legacy system only necessitates a few hundred lines of code. Another contribution of the paper is the application of this model to self-optimization of complex and multi-tier Internet services. Our experiments on an auction site show that Jade is able to maintain the client perceived performance stable when the workload varies, while, without Jade, the performance continuously decreases. Furthermore, in these experiments Jade allows twice as many client requests to be served as a system with no autonomic management.

KEY WORDS: Autonomic management, Legacy systems, Self-optimization, Internet services, Multi-tier architecture, J2EE

1 Introduction

1.1 Context

A common architectural organization for Internet applications is a multi-tier platform, in which each tier is a different legacy system that provides a specific function, e.g. a web tier which stores HTML pages, a business tier which executes the application business logic, and a database tier which stores non-ephemeral data. To ensure availability and high performance, these platforms are usually replicated on clusters of machines. The administration of such installations is an increasingly complex and error-prone task [17]. Autonomic computing, i.e. self-management in the face of evolving load conditions, hardware and software failures, and various forms of attacks, is an approach that aims at improving this situation [14]. Indeed, it provides a high-level support for deploying and configuring applications, which reduces configuration errors and human administrators' efforts. It allows the system, upon dynamic changes (e.g. performance variations, failures, etc.), to

perform the required reconfigurations without human intervention, thus saving administrators' time. It is also a means to save hardware resources, since these resources can be allocated on demand (as a reaction to a load peak or a failure) instead of being statically pre-allocated.

1.2 Challenges

Several challenges remain open in autonomic management, among which: (i) the management of legacy systems, (ii) the management of systems with complex architectures, and (iii) the management of systems with complex internal states.

Legacy systems. By a legacy system, we mean a black-box with an immutable interface, on which no assumptions may be made, e.g. a web server software, or an email server software. Usually, legacy system management solutions are implemented as ad-hoc solutions that are tied to a particular legacy system, e.g. an email server software [21], a web server software [15], or an application server software [7]. Unfortunately, with such an approach, autonomic management solutions need to be completely re-implemented each time a new legacy system is taken into account, even if the underlying management techniques are common to several legacy systems.

Complex architecture. The architecture of distributed systems, such as Internet services, is becoming more and more complex. Indedd, while the first generation of Internet services was based on the simple client/server architecture, the new generation of Internet services makes use of a series of servers connected together in a multi-tier architecture. With such an architecture, providing self-recovery upon a server failure for instance, does not simply imply restarting the failing server as an independent element, but also rebuiling the connections of that server with its front-end and back-end servers if any. This requires the knowledge of the global architecture of the system in order to perform reconfigurations of several elements of the system; such an architectural view is not provided by Internet services.

Complex internal state. Another type of system complexity is related to the nature of the internal state of the system. Systems such as web servers usually host static data (e.g. HTML files, images) that do not change over time; we call these systems static state systems. While systems such as database servers host dynamically changing data; we call such systems dynamic state systems. Let us consider, for instance, self-optimization through dynamic provisioning of new resources (i.e. servers) to face workload increases. If provisioning is applied to a web server, this would only require to start a new web server by initializing its static data. While if a new database server is provisioned, this would require to recompute the new state of the database before restarting it; and this is not trivial. Thus, self-optimization solutions usually do not provide dynamic resource provisioning for dynamic state systems [30, 10, 11].

1.3 Scientific contributions

The scientific contributions of this paper are to precisely address the above-mentioned issues through:

- The definition of an architectural framework for administrable components (i.e. Managed Elements, or MEs), which provides each element with a uniform management interface. Thus, any piece of managed legacy system is encapsulated in a well-defined software component called a wrapper. This provides the autonomic management environment a homogeneous view of the managed elements, even if the underlying elements are instances of different legacy systems.

- Furthermore, the architectural framework allows a system to be managed in a fine-grain way, as a dynamically evolvable assembly of elementary pieces. Thus, any piece of legacy software, as well as a hardware component, may be integrated in an assembly by binding wrappers of MEs together. This provides the autonomic management environment a general architectural view of the underlying managed system, even if that system has a complex architecture such as multi-tier platforms.

- The architectural framework also takes into account the internal state of the MEs in a system. More precisely, it integrates facilities to recreate static internal states as well as more complex states such as dynamic states.

We have designed and implemented Jade, an environment that provides autonomic management capabilities for distributed applications. Jade is composed of the above-mentioned architectural framework, in addition to an autonomic management framework. The autonomic management framework defines autonomous management actors (Autonomic Managers, or AMs), which use a control loop structure to apply an administration policy to a specific aspect (e.g. self-optimization, self-recovery, self-protection, etc.). Autonomic Managers administer systems that are made out of MEs. An Autonomic Manager may itself be equipped with a management interface, making it a ME, thus allowing hierarchical control. In this paper,

we focus on the self-optimization aspect of autonomic management, through dynamic resource provising to face workload variations [1].

We have evaluated this proposal using a clustered J2EE multi-tier Internet auction site modeled over eBay as a real life example. In our experiments, the workload was increased by a factor of 5, and then decreased by a factor of 5. In such a situation, Jade was able to maintain the web client perceived response time stable (around 590 ms in average). While the same experiments run without Jade result in a continuously increasing client response time (10.42 s in average). Moreover, our experiments show that Jade allows twice as many client requests to be served as a system with no autonomic management. On the other hand, the evaluation shows the benefits of the proposed approach in terms of flexibility and ease of use for the application designers and for the administrators. As an example of this flexibility, integrating a new managed legacy system only necessitates a few hundred lines of code, without requiring the whole autonomic management environment to be reimplemented.

1.4 Paper roadmap

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 our experimental environment, we made use of the Java 2 Platform, Enterprise Edition (J2EE) which defines a model for developing distributed applications, in a multi-tiered architecture, e.g. web applications [25]. Such applications usually start with requests from web clients that flow through an HTTP server front-end and 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 stores resources and data (see Figure 1).

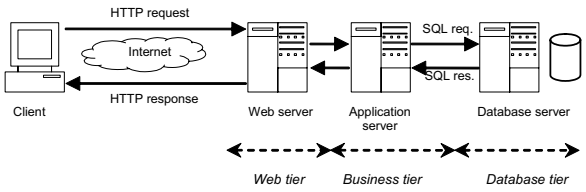

Figure 1. Architecture of dynamic web applications

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) [26]. 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 [12, 13]. This places a significant burden on servers [8]. 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 Internet service clustering solutions we can cite c-jdbc for a cluster of database

[1] Further details on the self-recovery aspect in Jade are given in [4].

servers [9], JBoss clustering for a cluster of JBoss EJB servers [6], mod_jk for a cluster of Tomcat Servlet servers [22], and the L4 switch for a cluster of replicated Apache web servers for example [24] (see Figure 2).

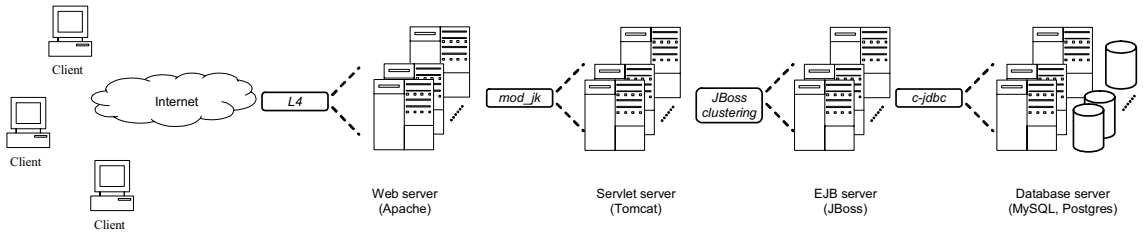

Figure 2. Internet services clusters

Thus, clustered multi-tier J2EE systems represent an interesting experimental environment for our autonomic management and self-optimization environment, since they bring together all the challenges described in section 1.2, namely:

- 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),
- the management of a distributed system with a complex global architecture, in the form of a multi-tier J2EE system,
- the management of systems with complex internal states, e.g. the J2EE database tier hosts dynamic data while the J2EE web tier hosts static data.

For all these reasons, we chose the J2EE platform as a candidate to illustrate the research contributions of this paper.

3 Design Principles and Architecture

Autonomic computing relies on the provision of control loops, in which an Autonomic Manager regulates parts of a system, called Managed Elements [14]. In order to be controllable, a Managed Element needs to be equipped with a management interface, which provides entry points for an Autonomic Manager. This interface should allow the manager to observe and to change the state of the element; the element may be an assembly of components, in which case the manager may modify the structure of this assembly.

We propose to use a software component model as a base for the design and implementation of Managed Elements. The component model that we use is Fractal [5], which has the following benefits: (i) it provides a uniform, adaptable control interface that allows introspection and dynamic binding of components; and (ii) it defines a hierarchical composition model for components, allowing a sub-component to be shared between enclosing components, at any level of granularity.

We use these properties to define a generic form of managed elements, providing a uniform management interface. This approach, Component Based Management (CBM), is further developed in Section 3.1 Section 3.2 describes the implementation of Managed Elements using CBM. Section 3.3 presents Autonomic Managers.

3.1 Component-based management

The CBM approach provides a management layer to a set of (legacy) hardware/software elements. An element, or set of elements, is wrapped in a Fractal component. This provides a means to:

- Managing legacy entities using a uniform model (the Fractal control interface), instead of relying on element-specific, hand-managed, configuration files.
- Managing complex environments with different points of view. For instance, using appropriate wrapping 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.

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

In the management layer, all components provide the same (uniform) management interface for the encapsulated elements, and the corresponding implementation is specific to each element (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 elements (ME). For instance, an administration program can inspect an Apache web server ME (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 ME, 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 next two sections present the application of component-based management to the implementation of managed elements and autonomic managers.

3.2 Implementing managed elements

Recall that a Managed Element is any entity (or group of entities) in the underlying system that needs self-management properties, e.g. self-optimization, self-recovery, etc. Instances of managed elements are hardware elements such as cluster nodes, or software elements such as middleware running on these nodes, or applications running on the middleware. Section 3.2.1 gives some details on the Fractal component model used to build a uniform management interface for the managed elements, and Section 3.2.2 illustrates the application of this technique to J2EE systems.

3.2.1 Managed element framework

In the Fractal model, a component is a run-time entity that has a distinct identity. The composition model is hierarchical, i.e. a component may contain other components, and subcomponents may be shared between enclosing components. In addition to its functional interfaces, a Fractal component has a set of control interfaces that provide management capabilities. These interfaces are implemented by controllers. The main available controllers are: (i) an attribute controller, which supports an interface to expose getter and setter methods for a component's attributes (i.e. configurable properties); (ii) a binding controller, which controls the connections between components; (iii) a lifecycle controller, which allows an explicit control over a component's execution (e.g. starting and stopping); and (iv) a content controller, which allows listing, adding and removing subcomponents in a composite component. These controllers provide a common base, from which element-specific controllers may be derived by extension. Some controllers may be irrelevant for a given class of elements.

3.2.2 J2EE managed elements

In our experiments of autonomic management of J2EE systems, we have implemented a set of Fractal components as wrappers of the managed elements for the different tiers of the J2EE architecture, (Apache web server, Tomcat Servlet server, and MySQL database server), and for a testbed J2EE application (the RUBiS auction site modeled over eBay). This application consists of static HTM pages, Java Servlet classes, and database tables, respectively deployed over the above servers. In this context, component bindings are used to represent the "horizontal" organization of J2EE systems, e.g. a front-end tier connected to a back-end tier; while composite components are used to represent the "vertical organization" of J2EE systems, i.e. a node, hosting a middleware tier, running an application (Figure 3).

We now give an example of a Fractal wrapper for the Aoache server managed element that is part of the J2EE architecture. The Managed Apache Middleware element wraps the Apache web server software, considered as a legacy software component. The wrapper provides an attribute controller, a binding controller and a lifecycle controller, which make up the management interface of Managed Apache:

The attribute controller interface is used to set attributes related to the local execution of the managed element. 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 elements. 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 Managed Elements, and provide the same management interface. These elements are then used by the J2EE self-optimization mechanism, as described in section 4.

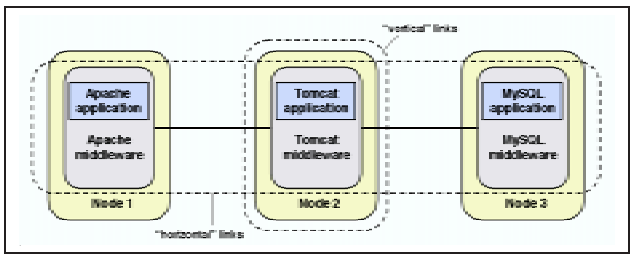

Figure 3. Example of a J2EE managed infrastructure

3.3 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 4 gives an overview of 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). In addition, the figure shows other services such as the Cluster Manager for the hardware elements of the platform which is used by the other two. A Software Installation Service implements a repository of software components and libraries needed for installing software on nodes when necessary. Each autonomic manager in Jade is based on a control loop that includes sensor, actuator and analysis/decision components.

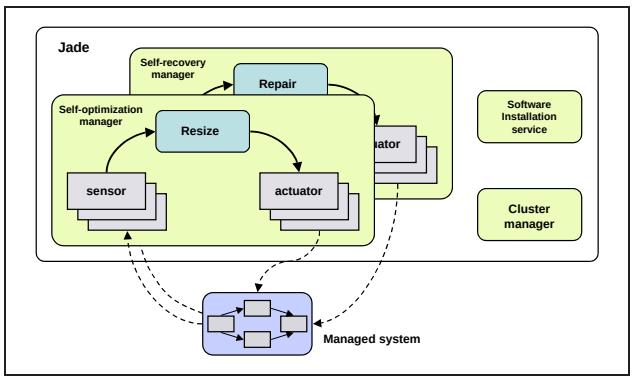

Figure 4. The Jade autonomic management system

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 for a self-recovery manager. Sensors must be efficient and lightweight. In the particular case of the self-optimization manager, sensors can 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 managed element in case of a self-recovery manager, or resizing the cluster of replicated elements 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 elements, updating connections between the tiers.

Thus, in Jade, wrapper components that encapsulate managed elements must provide Sensor and Actuator interfaces. Instances of Sensors and Actuators interfaces are, for a node hardware element, CPU/memory monitoring data as sensors, and the Managed Cluster operations as actuators, and for a web server software element, HTTP client response time monitoring data as a sensor, and Life-Cycle and Software Installation Service operations as actuators.

4 Implementation of Self-Optimization

In this section, we discuss the means of implementing self-optimization in replicated cluster-based systems, using the frameworks described in Section 3. Optimization is defined using two main criteria: first performance, as perceived by the clients (e.g. response time) or by the application's provider (e.g. global throughput); and second 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 task.

4.1 Self-sizeable elements

A standard pattern for element allocation in clustered servers is the load balancer. In this pattern, a given element R is statically replicated at deployment time and a front-end proxy P distributes incoming requests among the replicated servers.

Jade aims at autonomously adjusting the number of replicated elements used by the application when the load varies. The expected benefits are (i) improving resource utilization; and (ii) preserving user-perceived performance in the face of wide variations of the load. In this section, we concentrate on the architectural aspects of the proposed solution, i.e. how the Jade framework helps implementing dynamic allocation of the nodes. Technical details of the algorithms used, and experimental results, are reported in Section 5.

The pattern of a self-sizeable element (a form on autonomic manager) is based on a control loop, which comprises sensors, actuators, and a reactor.

The *sensors* periodically measure the chosen performance (or QoS) criteria, i.e. a combination of CPU usage and user-perceived response time.

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 elements of an application is implemented as adding or removing components in the application structure.

The *reactor* implements an analysis/decision algorithm. It receives notifications from sensors, and reacts, if needed, by increasing or decreasing the number of elements allocated to the controlled tier.

Figure 5 describes the generic pattern of a self-sizeable element. The main operations performed by the reactor when more elements are required are the foolowing: 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 elements with a dynamic data, and integrate the new replicas to the load balancer. Similarly, if some elements are under-utilized, the main operations performed by the reactor are the following: unbind some replicas from the load balancer, stop these replicas, and elease the nodes hosting these replicas if no longer used.

To allocate an additional element (i.e. node + software server), the reactor uses the services provided by Jade, e.g.é the Managed Cluster to allocate new nodes, the Software Installation Service to retrieve the necessary software elements.

In the following section, we describe in more detail the design and implementation of two self-optimization policies used to optimize, respectiveley, systems with static data and systems with dynamic data: a clueter of Servlets servers (Tomcat), and a cluster of database servers (MySQL).

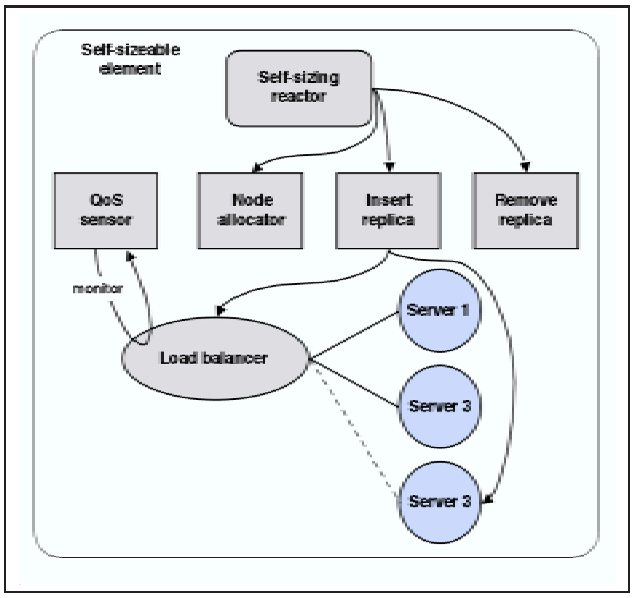

Figure 5. Self-optimization management in Jade

4.2 Self-sizeable static data elements

We consider a self-sizeable web container composed of a set of replicated Tomcat Servlet servers, configured as entities with static data (i.e. Servlet classes with no dynamically changing session information, etc.). Since the Tomcat servers are used here as static data entities, their replication does not entail inconsistency problem. The implementation of the self-sizeable web container has been realized by the implementation of all components involved in the control loop detailed previously.

Sensor. In the self-optimization manager, the sensor is a probe that collects CPU usage information on all the nodes that implement the web container. This probe computes a moving average of these data on each node in order to remove artifacts characterizing the CPU consumption of the web containers. It finally computes an average CPU load across all nodes, so as to observe a general load indication of the whole web container service.

Reactor. The information gathered by the sensors is processed by a reactor, which can trigger a reconfiguration when needed. Here, the decision logic implemented to trigger such a reconfiguration is based on thresholds on CPU loads. Details are provided in Section 5.

Actuator. The actuator implements the deployment and removal operations triggered by the reactor. These operations are easily realized thanks to the operations provided by the management interface. For example, to deploy a new Tomcat server, the system creates a new Tomcat component, inserts it into the self-sizeable web container component, binds it to the load-balancer component, and finally starts its execution.

4.3 Self-sizeable dynamic data elements

The self-sizeable database is realized as a cluster of database servers with a c-jdbc controller as a front-end; c-jdbc plays the role of a load balancer and a replication consistency manager [9], Here, each backend contains a full copy of the whole database (full mirroring).

The self-sizeable clustered database is implemented in a similar way. The three parts of the control loop have been specialized to the context of the sizing of a database:

Sensor. The sensors involved in the dynamic sizing of our clustered database component are identical to those used previously in the context of the self-sizeable web container. This CPU based information is also submitted to a moving average whose strength is adapted to the specificities of databases behaviors.

Reactor. The reactor's algorithm is based on two thresholds related to the CPU load. These thresholds are tuned so that they trigger reconfigurations when the elements are effectively overloaded or underloaded. More details in Section 4.

Actuator. The reconfigurations that are triggered for the self-sizeable database consist in deploying and removing one database backend in the clustered database. At the management interface level, these operations are identical to those described for the web container. However, these operations need to preserve the consistency of the state of the clustered database. This consistency issue is discussed below.

To manage a dynamic set of database back ends, a newly allocated back end 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 trace of all the requests that affect the state of the back-ends. Basically, all write requests are logged and indexed as strings in this recovery log. When a new backend is inserted in the clustered database, the state of this backend 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 backend to make it be up-to-date. Once these requests have been processed by the newly allocated backend, 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.

4.4 Generality of the approach

One of our objectives when building Jade was to provide a generic autonomic management environment which manages a variety of specific legacy systems. The current implementation of self-optimization management in Jade contains generic and specific parts. In fact, Jade is built on top of a set of generic bricks which are assembled in a specific way so as to provide self-optimization.

As a matter of fact, the current prototype is built using generic bricks implemented as components. 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 policiy 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.

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

Though the interface used to build the actuators is generic, the implementation of these actuators is contained in the wrappers and is therefore specific to the legacy components 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. Therefore we have built an adapted sensor providing an estimator of the CPU load (using a moving average). Therefore, this adapted sensor is specific to the optimization of system that we monitor even if it is built with generic components.

An evaluation of the generality of the proposed approach is given in Section 5.1.1

5 Evaluation

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

5.1 Qualitative evaluation

5.1.1 Generic environment

Recall that one of the objectives of our framework for autonomic management is the ability to manage a variety of legacy software environments, regardless of their specific interface and underlying implementation. This was made possible through the use of wrapping techniques in order to encapsulate legacy systems in Managed Element wrapper components. Therefore,

Jade consists of a set of generic sub-systems (e.g. Software Installation Service, Node Manager, Self-Optimization Manager), and a set of application-specific sub-systems (e.g. wrapper components for the MySQL database server, the Tomcat Servlet server). Table 1 gives the code size of Jade's generic subsystems and specific sub-systems. It provides a rough measure of the code factoring obtained thanks to the generic approach followed in Jade. Indeed, taking into account a new administered legacy system in Jade would require to implement a wrapper component that consists of, in average, 550 lines of Java code and a configuration file of 18 lines (this file describes the configuration of the system as a set of interconnected components, in the Fractal Architecture Definition Language). On the other hand, with ad-hoc (i.e. non-generic) implementations, taking into account a new legacy system would require to re-implement new versions of Autonomic Managers for that legacy system, for instance, a new Self-Optimization Manager and a new Self-Recovery Manager (with a total code size around 5 Klines of Java code).

		# Java classes	**Java code size**	**# ADL files**	**ADL file size**
Generic code	Software Installation Service	3	220 lines	1	16 lines
	Node Manager	12	1630 lines	13	250 lines
	Self-Optimization Manager	14	2340 lines	12	150 lines
	Total	***41***	***7240 lines***	***31***	***516 lines***
Specific code (i.e. wrapper components)	Tomcat Servlet server	3	550 lines	1	12 lines
	Tomcat load balancer	2	460 lines	1	14 lines
	MySQL database server	4	760 lines	3	40 lines
	c-jdbc database load balancer	1	810 lines	1	14 lines
	Rubis J2EE application	2	150 lines	1	11 lines
	Total	***11***	***2730 lines***	***7***	***91 lines***
	Average	***2***	***550 lines***	***1***	***18 lines***

Table 1. Generic code vs specific code in Jade

5.1.2 Dynamic and automatic reconfigurability

In this section, we show the benefits of using Jade to perform system reconfiguration, compared to an ad-hoc approach. Figure 6 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.

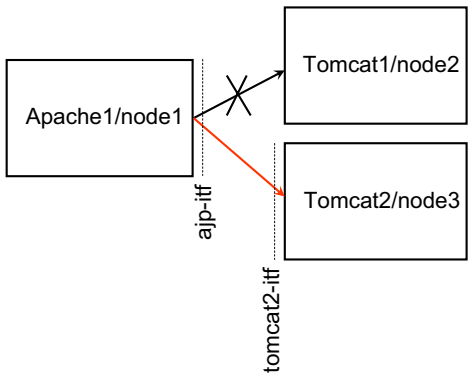

Figure 6. Reconfiguration scenario

Without the Jade infrastructure, this simple reconfiguration scenario requires the following steps to be manually done, in a legacy-dependent way: First log on node1, Then stop the Apache server by running the Apache *shutdown* script, And

edit and update the Apache configuration file to specifiy the configuration parameters of the new Tomcat server (Tomcat2 on node2) as follow:

```
worker.worker.port=8098
worker.worker.host=sci22
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.

While with Jade, the operations required to perform the same reconfiguration are simply few operations on the underlying managed elements, namely:

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

5.2 Quantitative evaluation

5.2.1 Evaluation environment

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. RUBiS comes with different mixes: a browsing mix in which clients execute 100% read-only requests and a bidding mix composed of 85% read-only interactions. 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 backends, 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 resiging of the underlying architecture. All the nodes are connected through a 100Mbps Ethernet LAN to form a cluster. Their hardware characteristics are given in Table 2.

Jade, Client Emulator, Application server load balancer	SMP Dual-Processor AMD Athlon 1800+ (1533 MHz), 1Gb RAM
Database load balancer	SMP Dual-Processor Intel Xeon 1800 MHz w. HyperThreading, 1Gb RAM
Database servers	Processor Intel Xeon 1800 Mhz, 1Gb RAM

Table 2. Hardware environment

Software Environment

The nodes run various versions of the Linux kernel: 2.4.20 for the database back-ends and the database and application load-balancer, 2.4.26 for Jade, and the client emulator, 2.6.12 for the web/application servers. The J2EE application has been deployed using open source middleware solutions: Jakarta Tomcat 3.3.2 [27] for the web and servlet servers (configured to handle static and dynamic pages), MySQL 4.0.17 [16] for the database back-ends, c-jdbc 2.0.2 [9] for the database load-balancer, PLB 0.3 [19] for the application server load-balancer. itemize 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, and IBM's JDK 1.4.2. We used the MySQL Connector/J 3.1.10 JDBC driver to connect the database load-balancer to the database back-ends.

5.2.2 Experimental results

We present here the experimental results obtained on the environment described above.

Performance Optimization

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.

(i) Workload

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). This workload is represented in Figure 7.

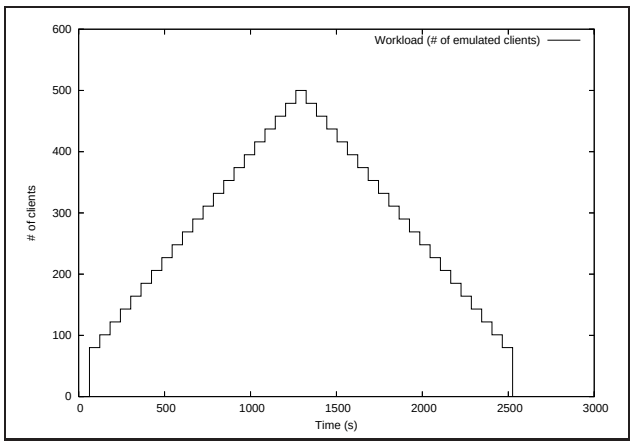

Figure 7. Emulated workload

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

(ii) Control loop

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. We now describe the operation of the control loops (see Figure 8). 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 60 seconds) 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 elements 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 elements 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 autonomic behavior of the managed system relies on the measured CPU usage exclusively. The choice of this sensor has been determined by experience. In fact, we have noticed that the CPU is the only bottleneck resource of the underlying experimental J2EE application.

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 back-ends.

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.

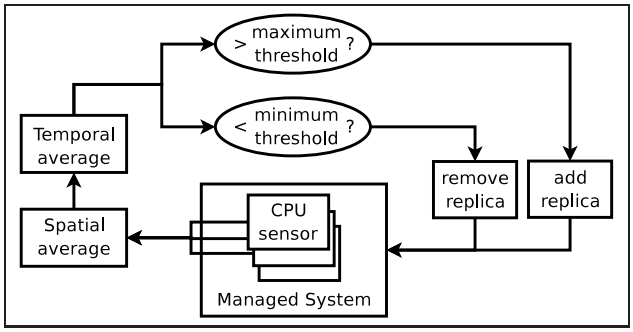

Figure 8. The control loop algorithm

Experimental results

The experimental results presented here are related to the above scenario. Figure 9 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 backend. The system now contains two database backends. 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 back-ends. 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 backend.

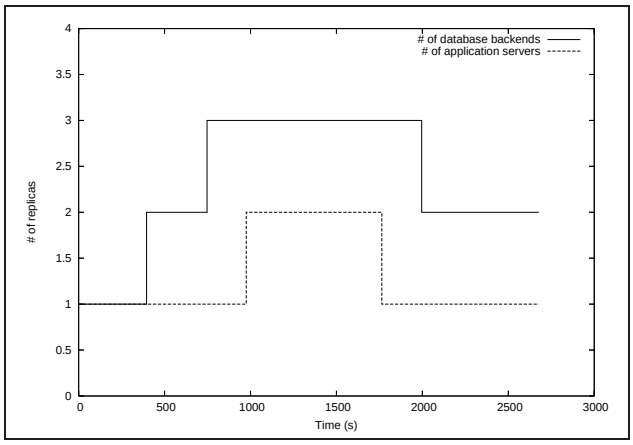

Figure 9. Dynamically adjusted number of replicas

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 10 and Figure 11 present the results of these experiments and show the thresholds used to trigger dynamic reconfigurations.

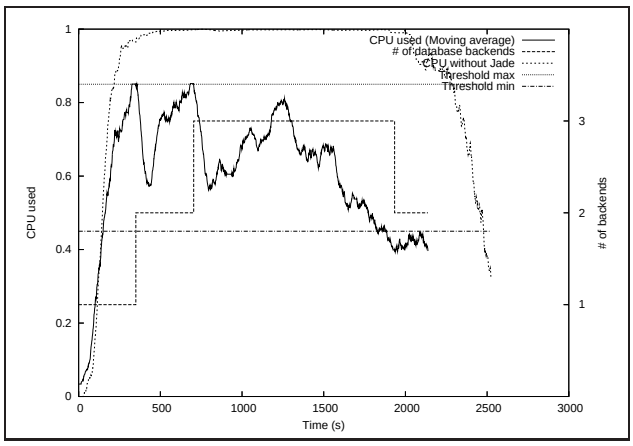

Figure 10. Behavior of the database tier

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 backend, 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 back-end. This behavior is quite visible in Figure 10. 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 trashing 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 trashing that affects the database. In fact, 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.

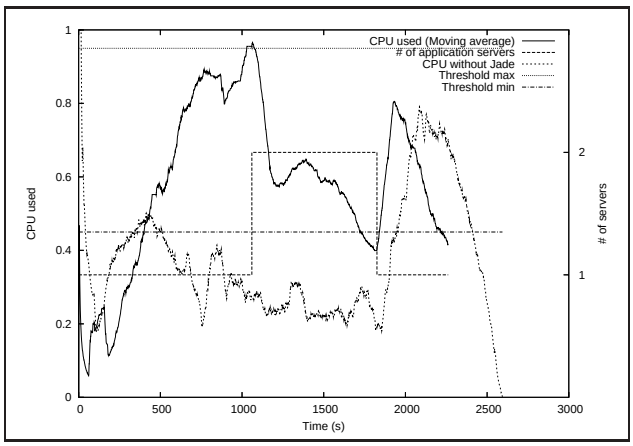

Figure 11. Behavior of the application tier

We notice that the statically configured application server generates higher CPU usage values at the end of the experiment even if the load decreases. This can be interpreted as a result of the end of the trashing of the database, which then returns results to the application server more promptly. This behavior is depicted in Figure 11.

In the following, we present the effect of the dynamic reconfiguration triggered by Jade in response to a detected performance loss. First, Table 3 gives performance indication of the managed system in terms of the total number of client requests processed during the whole experiment. Thus, the benefits of Jade autonomic management are clearly shown here.

Furthermore, when the J2EE application managed by Jade is dynamically sized to address performance issues, the number of processed requests is more than twice higher than the number of processed requests by the statically configured application.

	Total # of requests
With Jade	92512
Without Jade	41585

Table 3. Effect of the dynamic reconfigurations

We now consider the impact on performance in terms of client request response times. Figure 12 and Figure 13 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 experiments run without Jade result in a continuously increasing client response time (10.42 s in average) when the workload increases.

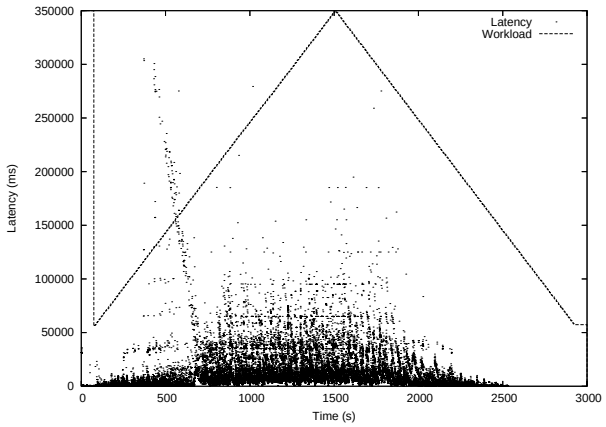

Figure 12. Response time without Jade

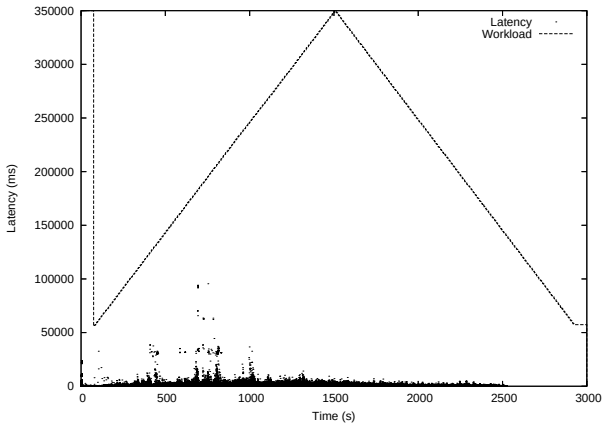

Figure 13. Response time with Jade

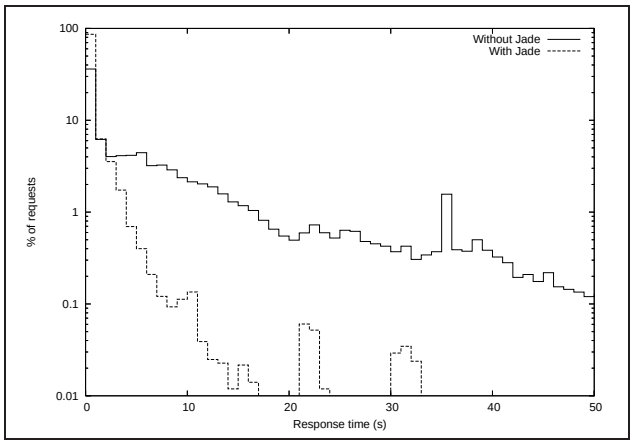

Figure 14. Distribution of the response time

Figure 14 gives another view of these results. It is a histogram that details the distribution of the response times among all the requests that were issued during the experiments with and without Jade. This figure shows that the proportion of the requests that are returned within one second reaches 86.25% when Jade is used, whereas without Jade, this proportion

is only 36.17%. Moreover, this also shows that the proportion of the requests that are returned within one and 20 seconds reaches only 13.47% with Jade, and 48.28% without Jade. Finally, while with Jade, no requests were returned in more than 40 seconds, still 5.5% of the requests processed by the statically configured system are returned in more than 40 seconds.

Performance overhead

The first results relate to the performance overhead induced by Jade on the managed J2EE application. This overhead 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 induced no dynamic reconfiguration. This way, we have observed the overhead induced by the presence of Jade management platform. This overhead is quantified in terms of average throughput, response time, processor and memory usage of the J2EE application in Figure 4.

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

Table 4. Performance overhead

The processor and memory usage are computed as average values over all cluster nodes implied 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 component (a Fractal Component Factory), which is 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, but only at the interfaces between large grained components.

5.2.3 Discussion

In the following, we briefly summarize the main results of the experiments with the self-sizing autonomic manager of Jade.

Firstly, the algorithm implemented by the control loop is simple, but proved quite effective (it may be compared with the load leveler algorithm used to prevent trashing in early multi-programmed systems). It can be seen as a base from which more refined algorithms can be elaborated. The controlled variable is the CPU load, which has been identified as the bottleneck for all tiers. Measurements of response time show that CPU load is well correlated to response time, for the testbed application. And finally, the autonomic manager entails a small perfomance penalty (of the order of 2 to 3% on response time for the experimental application).

However these main results lead us to develop more refined models and experimentations. Indeed, the reconfiguration algorithm can be specialized to optimize the managed system more accurately. A reconfiguration may then involve many replicas at a time. Moreover, our current experimentation exposes a single bottleneck resource (the CPU) but a realistic context may have various bottleneck resources under different circumstances. Therefore we plan to develop more sophisticated sensors which could be composed and correlated so as to provide relevant indicators to reconfigure the managed system. And finally, we expect to explore some other experimentation contexts such as grid-computing or embedded systems in which the reconfiguration scenarios and goals are different. As a matter of fact, a grid-computing related optimization algorithm would target the optimal usage of the resources.

6 Related Work

Several projects have addressed the issue of element management in a cluster of machines. For economical reasons, the cluster is supposed to be shared by a number of competing applications. Instead of statically allocating elements 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 [23] and DDSD [31]). Some of them (e.g. Cluster Reserves [3]

or Sharc [29]) 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 Oceano [2], QuID [20], OnCall [18], Cataclysm [28] or [30] fall into this category.

The Jade system described in this paper takes place in this second category. We assume a strong isolation between the applications hosted in the cluster, and the unit of resource allocation is an individual machine. However, the distinctive features of Jade are as follow.

Legacy systems. Jade aims at providing support for a variety of legacy systems. Indeed, with its underlying framework, Jade provides support for wrapping legacy systems into well-shaped components. These components exhibit a management interface which allows full configuration and reconfiguration of their properties and bindings. The experiments described in this paper takes place in the context of J2EE applications, involving the (re)configuration of complex software components such as Apache, Tomcat and MySQL servers. However, Jade is not tied to J2EE applications and can be used to manage many different software environments.

Complex distributed systems. Our experiments with self-optimization of multi-tier J2EE architectures show that Jade is able to manage complex distributed systems with a series of servers, while most of the previous solutions only target the simple client-server architecture.

Complex state. Our experiments with self-sizing in J2EE systems show that Jade can manage components with an internal state that consists of static data, or an internal state that consists of dynamic data. In both cases, the managed components may be replicated on several nodes and kept consistent. Our experiment with the J2EE database tier (dynamic data component) as well as the J2EE web tier (static data components) show that Jade is able to automatically manage the synchronization of the replicas when needed. In contrast, most of previous work relied on simple application workload with static data only (e.g. HTTP servers), which do not have propose synchronization solutions.

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 a set of architectural frameworks for constructing flexible and efficient autonomic management facilities for such systems. To prove the validity of our approach, we have applied these frameworks to the self-optimization of J2EE applications in the face of the wide load variations observed in Internet applications

Jade consists of two main frameworks: a framework for administrable components, which provides the administered system's components with a uniform management interface; and a framework for autonomic managers, which allows regulating a set of managed components for a specific management aspect.

The construction of these frameworks relies on the Component-Based Management approach (CBM), using a simple and generic architectural model to manage any hardware or legacy software infrastructure. The underlying component model provides the introspection and dynamic control facilities needed for the provision of a management interface to administrable elements. Autonomic managers use the component abstraction to reconfigure the underlying infrastructure and can discover the overall system infrastructure.

Providing a generic framework for autonomic administration proves beneficial, since a large proportion of the code implementing management functions is itself generic (more than 70% in our experiments). The uniform management structure also greatly simplifies the administrators' task. At the component level, adding or removing a servlet server component is done in the same way as adding or removing a database. Autonomic managers also hide the complexity of heterogeneous configuration files.

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 platform. 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. 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)*, Sant 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] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. A Microrebootable System: Design, Implementation, and Evaluation. In *6th Symposium on Operating Systems Design and Implementation (OSDI-2004)*, San Francisco, CA, Dec. 2004.

[8] 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.

[9] 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/.

[10] J. Chase, D. Anderson, P. Thakar, A. Vahdat, and R. Doyle. Managing Energy and Server Resources in Hosting Centers. In *18th Symposium on Operating Systems Principles (SOSP-2001)*, Chateau Lake Louise, Banff, Canada, Oct. 2001.

[11] R. Doyle, J. Chase, O. Asad, W. Jin, and A. Vahdat. Model-Based Resource Provisioning in a Web Service Utility. In *4th USENIX Symposium on Internet Technologies and Systems (USITS-2003)*, Seattle, WA, Mar. 2003.

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

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

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

[15] M. Luo and C. S. Yang. Constructing Zero-Loss Web Services. In *20th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM-2001)*, Anchorage, AL, Apr. 2001.

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

[17] K. Nagaraja, F. Oliveira, R. Bianchini, R. P. Martin, and T. D. Nguyen. Understanding and Dealing with Operator Mistakes in Internet Services. In *6th Symposium on Operating System Design and Implementation (OSDI-2004)*, San Francisco, CA, Dec. 2004.

[18] 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.

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

[20] 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.

[21] Y. Saito, B. Bershad, and H. Levy. Manageability, Availability and Performance in Porcupine: A Highly Scalable, Cluster-Based Mail Service. *ACM Transactions on Computer Systems*, 18, Aug. 2000.

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

[23] 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.

[24] 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.

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

[26] Sun Microsystems. Java DataBase Connection (JDBC). http://java.sun.com/jdbc/.
[27] The Apache Software Foundation. Apache Tomcat. http://tomcat.apache.org/.
[28] B. Urgaonkar and P. Shenoy. Cataclysm: Handling Extreme Overloads in Internet Services. Technical report, Department of Computer Science, University of Massachusetts, Nov. 2004.
[29] B. Urgaonkar and P. Shenoy. Sharc: Managing CPU and network bandwidth in shared clusters. *IEEE Transactions on Parallel and Distributed Systems*, 15(1), 2004.
[30] 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.
[31] 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.