Delivery Selfware SP3

Lot 3, December 18, 2008

Autonomic Management of J2EE Servers

Authors

Loris Bouzonnet (Bull S.A.S.)
Fabienne Boyer (INRIA/Sardes)
Marc Léger (FT/EMN)
Nicolas Loriant (EMN)
Benoît Pelletier (Bull S.A.S.)
Sylvain Sicard (INRIA/Sardes)
Christophe Taton (INRIA/Sardes)
Bruno Dillenseger (France Telecom)

Version 2.0

Contents

1 Introduction

Autonomic computing, which aims at the construction of self-managing and self-adapting computer systems, has emerged as an important goal for many actors in the domain of large scale distributed environments and applications. Indeed, such environments and applications are becoming increasingly sophisticated, involving numerous complex software developed with heterogeneous programming models. The main difficulty raised by this situation concerns the management of the environment and its applications (installation, configuration, tuning, repair ...), that often relies on several proprietary configuration facilities.

One approach to autonomic computing, called the *control approach*, views the functioning of an autonomic computing system as a feedback control loop. As presented in the SP1-L1 document, the Selfware platform follows this approach, by providing systems and applications with self-management capabilities, including self-configuration (automatic configuration according to a specified policy), Self-Optimization (continuous performance monitoring), Self-Repair (detecting defects and failures, and taking corrective actions), and self-protection (taking preventive measures and defending against malicious attacks).

This platform has been used to manage clustered J2EE application servers, by applying autonomic repair and optimization control loops. The objective of this document is to give details on these experiences. The following sub-sections firstly recall the main design principles of the Selfware platform, and then describe the different control loops implementations in two environments J2EE 1.4 and Java EE 5.

2 Selfware Design Principles

This section recalls the main design principles of the Selfware platform, based on the notions of Managed Elements and Autonomic Managers.

2.1 Managed Elements and Autonomic Managers

As detailed in the SP1-L1 document [1], the autonomic regulation provided by the Selfware infrastructure on a managed system is based on managed elements (ME) and autonomic managers (AM). A system managed with Selfware is more precisely constituted by a collection of managed elements, that may consist of a single elementary hardware of software element, or may be a complex system in itself, such as a clustered application server.

A managed element provides sensor and actuator interfaces respectively allowing to observe and manipulate it. Sensor and actuator interfaces are used by autonomic managers, that regulate a managed system through feedback control loops, as illustrated in Figure 1. An autonomic element is the ensemble including a set of managed elements controlled by autonomic managers.

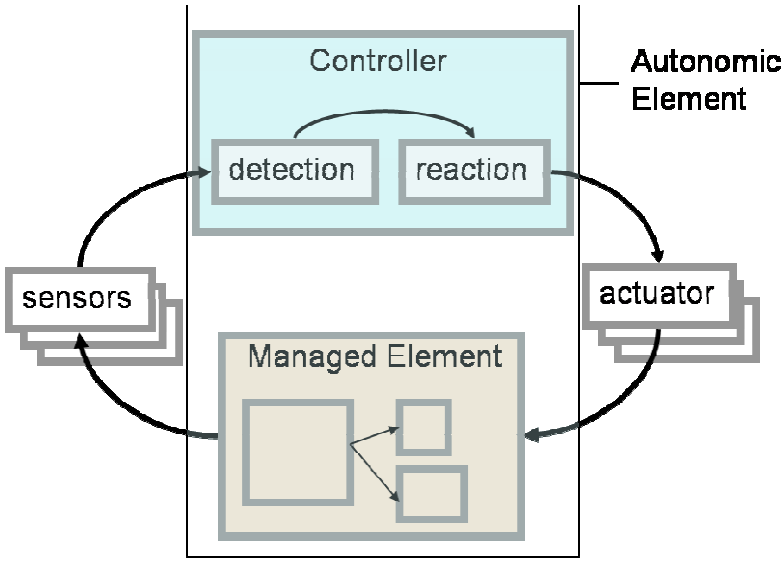

Figure 1. An autonomic element

A main design choice is to rely on a component model for building both Managed Elements and Autonomic Managers. The component model we use is Fractal [2]. A managed element is implemented as a Fractal component that encapsulates a controlled legacy entity. In the same way, an autonomic manager is a Fractal component that monitors a set of managed elements, analyzes notifications coming from managed elements sensors, diagnoses the state of the system, decides on a plan of actions and finally, executes the corresponding command plan.

2.2 Autonomic managers

Autonomic Managers administer legacy systems encapsulated in Managed Elements. Figure 2 illustrates the general architecture of the Selfware framework. In accordance with the purpose of this document, we recall the main principles of two autonomic managers: the Self-Repair manager and the Self-Optimization manager. These managers use some common services provided by the Selfware platform, that are detailed in the SP1-L2 document [2], and summarized in the following sub-section.

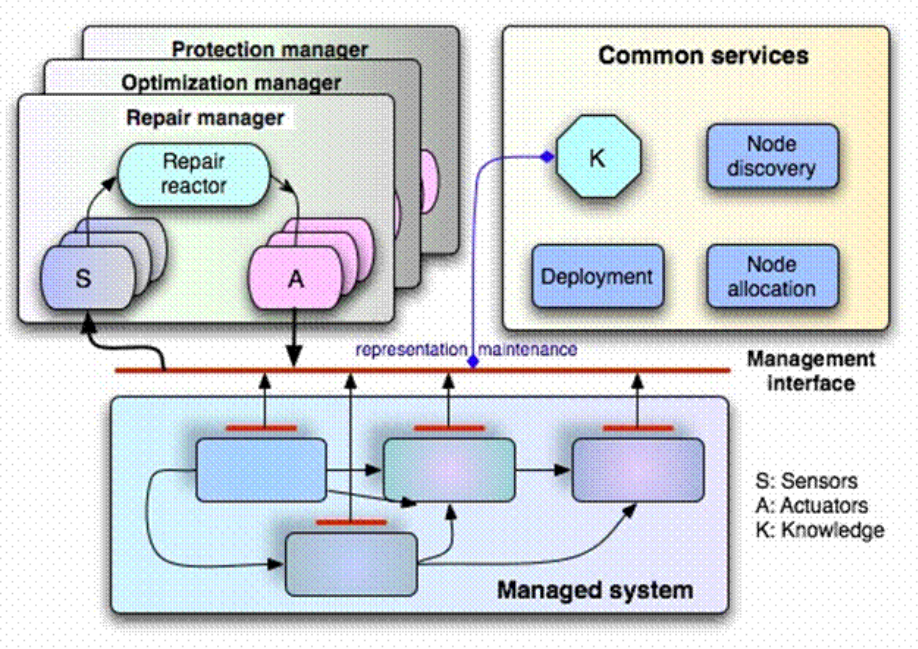

Figure 2.The Selfware global architecture

2.2.1 Main common Services

The three main common services used by the Self-Repair and self-optimizer managers are the Node Allocation Service, the Deployment Service and the System Representation Service.

The Node Allocation Service implements cluster resource reservation by allowing to allocate/de-allocate nodes in a cluster. The Software Installation Service allows to install a software on a node if not already existing. This service implements a repository that stores software libraries needed by managed legacy systems, but also the libraries of the wrappers associated with these legacy systems. This service is implemented using the OSGi technology, and the stored software libraries are OSGi bundles (i.e. specific Java libraries, see http://www.osgi.org/).

The Deployment Service allows to create and start Managed Elements (i.e. legacy systems and their wrapper components) on the allocated nodes. Moreover, deployment operations may automatically induce software installation operations if the necessary software is not already installed. Selfware Deployment Service is built on top of Fractal deployment features. Furthermore, it is important to notice that the Software Installation Service and Deployment Service are generic and may apply to any legacy system.

Finally, the Selfware System Representation Service provides a checkpoint of the Managed Elements. This is mainly used by the Self-Repair autonomic manager upon failures of MEs, while the other Selfware services may be used by all Autonomic Managers.

2.2.2 The Self-Repair autonomic manager

The Self-Repair Manager deals with fail-stop failures of MEs, e.g. a node hardware failure, or a server (i.e. middleware) crash. This manager allows MEs to recover from their failures by periodically monitoring their status via heart-beat sensors. When a failure of a ME is detected, the repair algorithm consists in first accessing the System Representation Service to retrieve the state of the failed ME prior to failure. The System Representation Service is particularly necessary in case the failing ME is a node, which results in loosing the wrapper associated with that ME, and thus being unable to introspect that wrapper to retrieve the state of the failed ME prior to failure. The state of the failed ME retrieved from the System Representation Service is then used to rebuild the failed legacy element as it was prior to failure.

To this purpose, new nodes are allocated from the Node Reservation Service if necessary, legacy software elements are installed if needed using the Software Installation Service, and a new ME is redeployed to replace the failing one using the Deployment Service. Furthermore, if the failed ME was bound to other MEs, the bindings are recreated with the recovered ME to reflect the same connections. And if the failed ME contained sub-MEs (i.e. sub-components), these sub-MEs are recursively repaired.

Furthermore, it is important to notice that the Self-Repair Manager is replicated (see SP2-L1 [3]) in order to prevent the manager from being a single point of failure, and thus guarantying fault-tolerance continuity.

2.2.3 The Self-Optimization autonomic manager

The Self-Optimization Managers aim at maximizing application performance while minimizing the underlying resource usage (e.g. cluster nodes) through dynamic resource provisioning and dynamic load-balancing. This kind of managers target MEs that represent clusters of replicated MEs, e.g. a cluster of replicated web servers or a cluster of ejb containers. The cluster ME is periodically monitored via load sensors (e.g. CPU load, network activity, memory consumption, or an aggregation of sensors). We describe hereinafter the two kinds of Self-Optimization Managers:

2.2.3.1 Dynamic resource provisioning

When the load exceeds a given maximum threshold, the cluster ME is resized by dynamically adding new replicas as sub-MEs of the cluster ME. This consists in first contacting the Node Allocation Service to allocate a new node, then using the Software Installation Service to install legacy software elements if necessary, and introspecting one of the sub-MEs of the cluster ME to replicate it on the new node as a newly deployed sub-ME. Symmetrically, if the overall load of a cluster ME is below a given minimum threshold, that means that the underlying cluster nodes are under-utilized. Thus, the cluster ME is dynamically resized by removing one or more of its replicated sub-MEs, and de-allocating the underlying nodes if no more used.

Furthermore, if the resized cluster ME consists of stateful replicated MEs with a dynamically changing state, replica consistency must be ensured when resizing the cluster ME; this is typically based on consistency policies underlying the legacy MEs. Replica consistency is supposed to be ensured through the underlying database clustering system.

2.2.3.2 Dynamic load-balancing

When a cluster is not fully congested, i.e. if a few nodes are not yet overloaded, load-balancing parameters tuning can be a less costly approach to perform optimizations. The purpose of this manager is to maximize the use of available resources in a cluster, by adding load-balancers as managed elements of client-side.

Each ME is instrumented (sensors and actuators) and a load-factor is computed for each replicated ME. A global load-factor for the cluster ME gives a health indicator for the cluster ME. When the system can be optimized without collapsing the cluster, the load-balancer can be tuned for routing the requests towards the lowest loaded replicated ME. The load-balancer implements a weighted round robin algorithm and provides an interface for setting dynamically new weights for each replica. The autonomic manager aims at tuning these parameters according the current load of each replicated ME.

3 Autonomic management of a J2EE Application Server

The Selfware platform has been used to manage the JOnAS J2EE application server in a clustered environment. In the following, we first provide background information on this application server and motivate its usage as a validation environment, before describing its self-management with Selfware.

3.1 Background and motivations

Nowadays, a large portion of web applications follow a multi-tier architecture. Java 2 Platform, Enterprise Edition (J2EE) defines the Java standard for developing multi-tier applications [27]. 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 enterprise server to execute the business logic of the application and generate web pages on-the-fly, and finally to a database that stores non-ephemeral data. However, the complexity of multi-tier architectures and their low rate for delivering dynamic web documents (often one or two orders of magnitudes slower than static documents) place a significant burden on servers [17]. To face high loads and provide higher service scalability, a commonly used approach is the replication of servers in clusters.

Replication-based clustering solutions are responsible of dynamically balancing the load among replicas, and managing replica consistency if any (e.g. database replica consistency). Instances of J2EE clustering solutions are c-jdbc for a cluster of database servers [11], JOnAS clustering for a cluster of JOnAS EJB servers [29], Tomcat clustering for a cluster of Tomcat Servlet servers [28], and the L4 switch for a cluster of replicated Apache web servers for example.

In this context, including autonomic management to clustered multi-tier web systems brings the following interesting challenges:

- The management of a variety of legacy systems in a generic way, since each tier in the multi-tier architecture embeds a different piece of software (e.g. a web server, an enterprise server, or a database server).
- The management of distributed systems with complex architectures in a generic way, where in addition to the multi-tier organization, each tier is a software stack that may be replicated.

3.2 Java environments, applications and scenarios

The Selfware platform supports two Java environments:

- An environment J2EE 1.4, for *Java 2 Platform, Enterprise Edition*
- An environment Java EE 5, for *Java Platform, Enterprise Edition*

J2EE and Java EE design two consecutive generations of platform for server programming in the Java programming language. The last version number of J2EE is 1.4 and the first one of Java EE is 5. These platforms are defined by specifications formalized by two Java Community Processes: JSR 151for J2EE 1.4 and JSR 244 for Java EE 5.

Two scenarios use the J2EE 1.4 environment and an other one uses the environment Java EE 5. According to the scenarios, different applications have been used. The next sub-sections will describe them and will map a Java environment and an application for each scenario.

3.2.1 The J2EE application Rubis

We considered a J2EE multi-tier web system consisting of three tiers: a web tier as a front-end, an enterprise tier as a middle- tier, and a database tier as a back-end. A first node hosts the Apache web server middleware [6], a second node hosts the Tomcat enterprise server middleware [28], and a third node hosts the MySQL database server middleware [20].

This multi-tier system runs the Rubis e-commerce application which models an auction site [5]. More precisely, Rubis' web documents are deployed on the Apache web middleware, where they represent the application layer of that tier. Rubis' Java Servlets are deployed on the Tomcat enterprise middleware, where they represent the application layer running on that middleware. And Rubis' database tables are deployed on the MySQL database middleware where they represent the application layer of that tier. Moreover, each tier is replicated to form a highly scalable system; the PLB system is used as the web tier clustering solution [23], the Tomcat clustering system is used to replicate the enterprise tier [28], and the c-jdbc database clustering solution is used to replicate the database tier [11]. Thus, this system brings together 9 different pieces of legacy software, namely the Apache middleware, Tomcat middleware, the MySQL middleware, the Rubis web application, the Rubis enterprise application, the Rubis database application, the PLB web clustering system, the Tomcat enterprise clustering system, and the c-jdbc database clustering system.

This system exhibits a complex architecture consisting of a multi-tier system representing a series of three clusters; each cluster is a collection of replicated systems; and each replicated system is a stack of node/middleware/application. We used Selfware in order to integrate self-management properties to such a system.

To this end, we built a Selfware wrapper for each legacy system, to obtain a total of 6 specific legacy Managed Elements corresponding to the six middleware and application elements.

3.2.2 The J2EE application SOAPSOO

SOAPSOO is a web application based on J2EE technologies from France Telecom's information system. It manages articles of two kinds (hardware and service), associated to catalogs and contracts. Through a web interface, users may browse, consult, modify, create or delete articles, catalogs and contracts (see Figure 3).

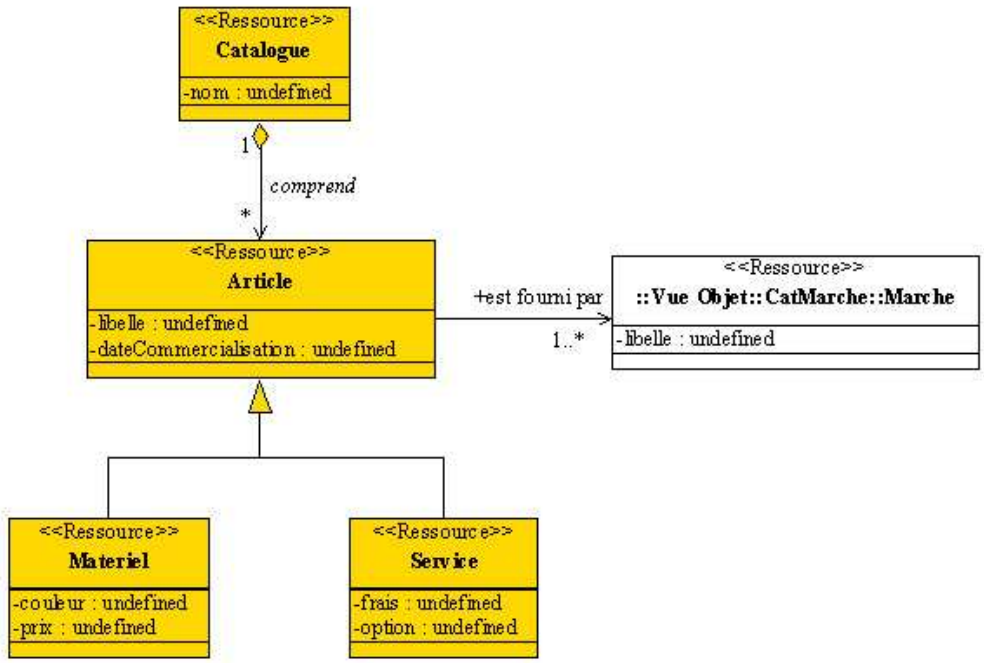

Figure 3. Data model of the SOAPSOO on-line catalog web application (simplified view)

Connection to the application is done through a usual authentication page with a login identifier and a password. Authentication is based on an HTTP POST method. Then, further user interactions are based on HTTP GET methods only. The presentation tier is implemented on Struts. It uses the Axis middleware to call the business logic through the SOAP protocol over HTTP. The persistence

tier is achieved on top of a relational database through the Data Access Object standard using the Object Relational Mapping (DAO ORM).

3.2.3 The scenarios

The used Java environment and application, for each scenario, are given by the Figure 4.

Scenario	Java environment	Application
Self-repair & self-optimization (§3.3)	J2EE 1.4	Rubis
Self-repair with integrity constraints ($3.4)	J2EE 1.4	SOAPSOO
Self-benchmarking & Self-optimization ($3.5)	Java EE 5	SOAPSOO

Figure 4.Java environment and application for each scenario

The scenarios will be describes in the next sections.

3.3 Self-repair and self-optimization of J2EE 1.4 Application Server

3.3.1 Experimental environment

The evaluation has been realized with the Rubis multi-tier J2EE application benchmark which implements an auction site [5]. Rubis defines several web interactions (e.g. registering new users, browsing, buying or selling items); and it provides a benchmarking tool that emulates web client behaviours and generates a tuneable workload. Rubis comes with two 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 application. Rubis was deployed as a cluster-based replicated multi-tier system, consisting of a cluster of replicated web/enterprise servers as a front-end, and a cluster of replicated database servers as a backend. We used the Rubis 1.4.2 version of the multi-tier J2EE application running on several middleware platforms: Apache 1.3.29 as a web server [2], Jakarta Tomcat 3.3.2 as an enterprise server [28], MySQL 4.0.17 as a database server [20], PLB 0.3 as the web server clustering solution [23], Tomcat clustering as the enterprise server clustering solution [28], and c-jdbc 2.0.2 as the database server clustering system [11]. Experiments were performed on the Linux kernel running x86-compatible machines, with 1GB RAM and 1800MHz, connected via a 100Mb/s Ethernet LAN to form a cluster.

3.3.2 Generic Approach

One of the objectives of the Selfware autonomic management framework is to manage a variety of legacy software systems, regardless of their specific interface and underlying implementation. Another objective is to deal with different system architectures, as complex as cluster-based multi-tier architectures. To this end, the Selfware framework consists of a set of generic common services. In addition to these generic services, we built application-specific sub-systems that are needed in a particular application domain (e.g. Selfware wrappers for the different legacy Managed Elements involved in a multi-tier web system).

		# Java classes	Java code size	# ADL files	ADL file size
Generic code	Software Installation Service	3	430 lines	9	830 lines
	Deployment Service	21	2600 lines	9	830 lines
	Node Reservation Service	2	475 lines	1	30 lines
	System Representation Service	40	6630 lines	–	–
	Self-Recovery Manager	48	4750 lines	26	430 lines
	Self-Optimization Manager	14	2340 lines	12	150 lines
	Architectural MEs	4	840 lines	4	200 lines
	Total	***132***	***18065 lines***	***61***	***2470 lines***
Specific code	Rubis web application	1	150 lines	1	11 lines
	Rubis enterprise application	2	150 lines	1	11 lines
	Rubis database application	1	150 lines	1	11 lines
	Apache web middleware	3	800 lines	1	16 lines
	Tomcat enterprise middleware	3	550 lines	1	12 lines
	MySQL database middleware	4	760 lines	3	40 lines
	PLB web clustring	2	460 lines	1	14 lines
	Tomcat enterprise clustring	2	460 lines	1	14 lines
	c-jdbc database clustering	1	810 lines	1	14 lines
	Total	***19***	***4290 lines***	***11***	***143 lines***
	Average	***2***	***477 lines***	***1***	***16 lines***

Figure 5. Code size of Selfware's generic services and specific sub-systems

Figure 5 gives the code size of Selfware's generic services and specific sub-systems. It provides a rough measure of the code factoring obtained thanks to the generic approach followed in Selfware. Indeed, taking into account a new administered legacy system in Selfware would require to implement a Selfware wrapper that consists of, in average, 477 lines of Java code and a Fractal configuration file of 16 lines (i.e. Fractal ADL). On the other hand, with an ad-hoc (i.e. non-generic approach), taking into account a new legacy system would require to re-implement new versions of Autonomic Managers for that legacy system, for instance, a new Self-Optimization Manager and a new Self-Repair Manager (with a total code size around 7 Klines of Java code).

3.3.3 Self-Repair

The first Self-Repair experiments were performed on a three-tier auction site consisting of a web server, an enterprise server and a database server, with a medium workload of 300 web clients. If no Self-Repair underlies the system, when a failure occurs on the web server, the auction site becomes unavailable. Thus, all new client requests result in an HTTP error until the end of the experiment. While when Selfware is used, the failure is automatically repaired by replacing the failed web server by a new one, and thus guarantying service continuity.

We run other experiments on the multi-tier auction site consisting of a web server, a cluster of two replicated enterprise servers and a database server. Here, a failure occurs on an enterprise server at time 195 seconds (see Figure 7). Without Selfware, the enterprise server clustering solution applies fail-over techniques to provide global service availability. However, this is obtained at the expense of service performance, where the load of the failed enterprise server is moved to the remaining replica (the CPU usage of the latter grows from approximately 25% to 50%). While when Selfware is used, in addition to service continuity, automatic recovery ensures performance stability.

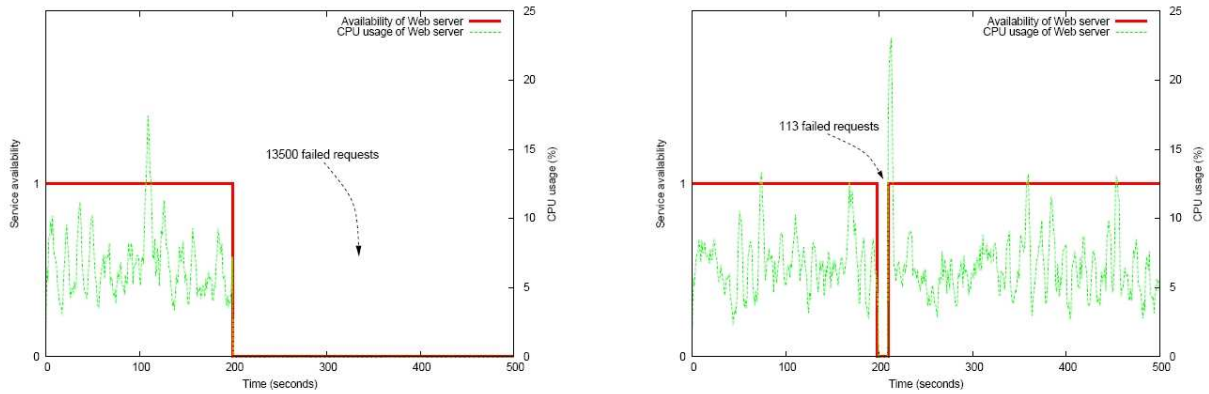

Figure 6. Web server behavior in presence of failures

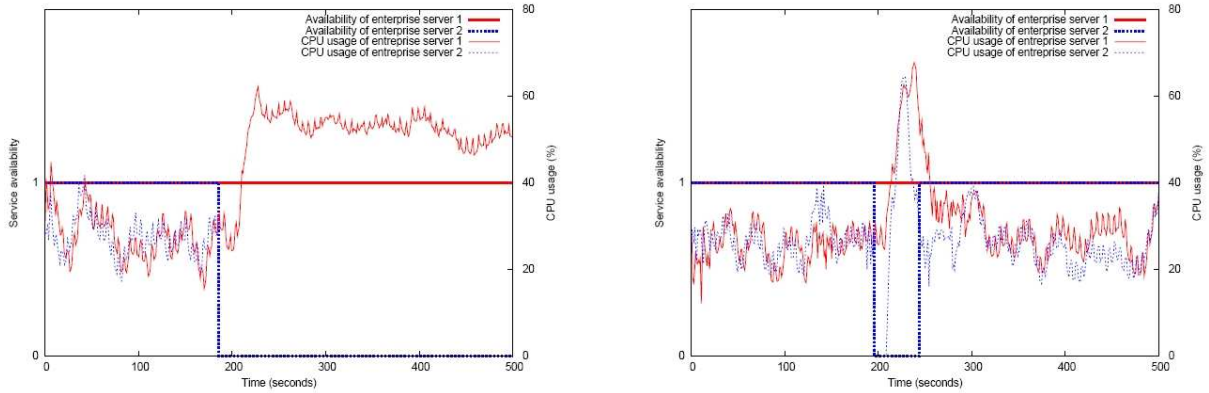

Figure 7. Replicated enterprise servers behavior in presence of failures

3.3.4 Self-Optimization

In order to evaluate the Self-Optimization policy provided by Selfware, we considered a scenario where the application workload varies dynamically. At the beginning of the experiment, the web application is submitted to a medium workload (80 clients); then the load increases progressively up to 500 clients; and finally the load decreases symmetrically down to reach 80 clients.

Initially, the multi-tier auction site is deployed on one enterprise (and web) front-end server and one database back-end server. Since CPU is the only bottleneck resource in these experiments, the managed system elements (i.e. enterprise tier and database tier) are monitored by sensors that gather CPU usage information every second and compute a spatial 1 and temporal 2 average CPU usage value. The Self-Optimization manager ensures that the average CPU usage is kept between a minimum and a maximum thresholds. In order to prevent oscillations due to parallel reconfigurations started on the front-end tier and the back-end tier of the multi- tier managed system, the reconfiguration workflow associated with the underlying Multi-Tier Managed Element specifies that a reconfiguration started on one of the tiers inhibits any new reconfiguration for a short period.

Figure 8 shows the variation of the number of replicas, for both the enterprise servers and database servers when the application workload varies. As the workload progressively increases, the average resource consumption of the cluster of replicated database systems also increases, and this tier

[1] Over nodes of replicated elements
[2] Over the last 60 seconds

becomes a bottleneck. An allocation of a new database replica is triggered, which results in a clustered back-end containing two database systems. The workload continues growing and triggers another node allocation for the clustered database. The workload increases further; and this places the bottleneck on the front-end tier. An allocation of a new enterprise is triggered, resulting in a system composed of two enterprise systems and three database systems. The workload then increases without saturating this configuration before it starts decreasing. This workload decrease implies a decrease of the resource consumption of the front-end tier which triggers a de-allocation of one its replicas, and then a low resource consumption of the clustered database, which triggers a de-allocation of a database replica.

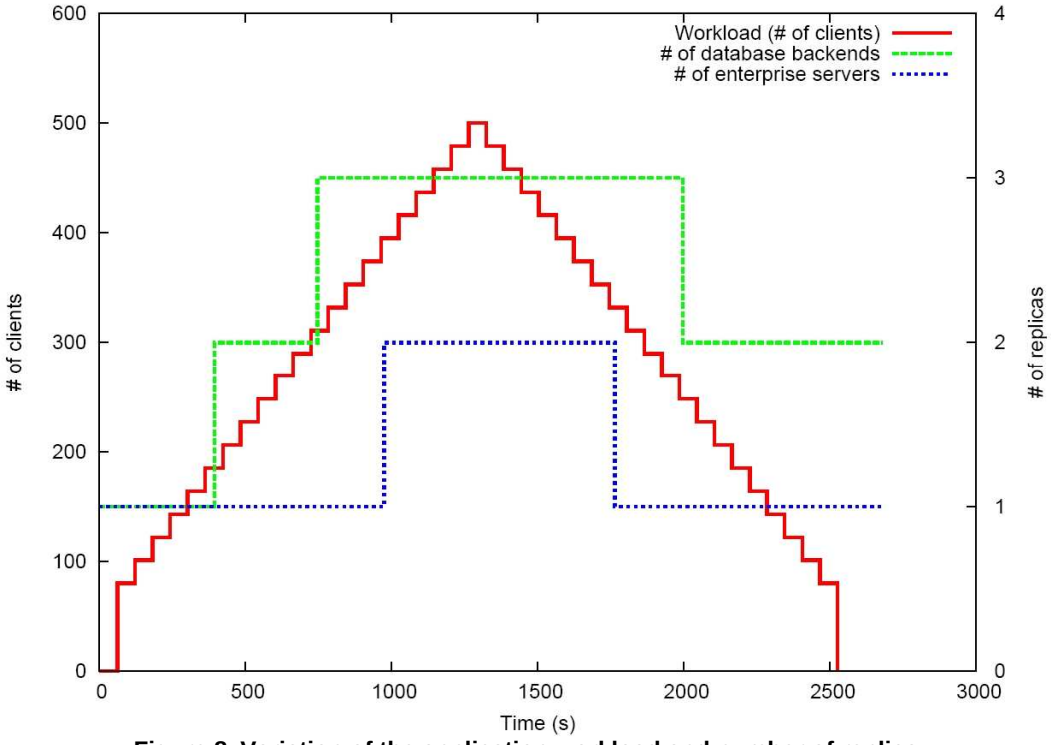

Figure 8. Variation of the application workload and number of replica

These experiments were run, on the one hand, on a system managed with Selfware and, on the other hand, on the same system with no Self-Optimization. Figure 9 presents the experimental results with the applied thresholds. In fFigure 9 -(a), when the CPU usage reaches the maximum threshold, a new database replica is deployed, which implies a decrease of the CPU usage. Symmetrically, when the CPU usage gets under the minimum threshold, a back-end replica is removed. In contrast, when the system is not self-managed, as the workload increases, the CPU usage eventually saturates. This results in a trashing of the database, which stops when the load decreases. Figure 9-(b) presents similar results related to the front-end tier. However, when the system is not self-managed, its behavior should be correlated with the trashing that affects the database. Indeed, since the database is already saturated, the enterprise server spends most of its time waiting for the database responses. This explains why the CPU usage measured during high loads remains moderate. Furthermore, we notice that the non-self-managed enterprise 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 front-end server more promptly.

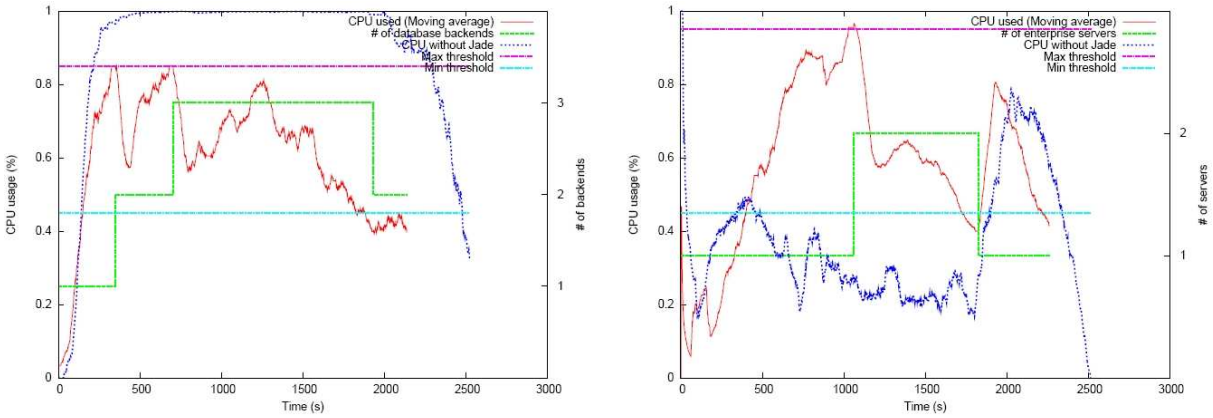

Figure 9. Behavior upon workload variations

On the other hand, it is important to notice that, unlike most of the existing autonomic management proposals [31][12][15], Selfware was able to manage systems with replicated static data (enterprise replicas in our experiments) as well as systems with replicated dynamically changing data (database replicas). Here, replica consistency was based on underlying clustering solutions, e.g. enterprise server clustering and database server clustering.

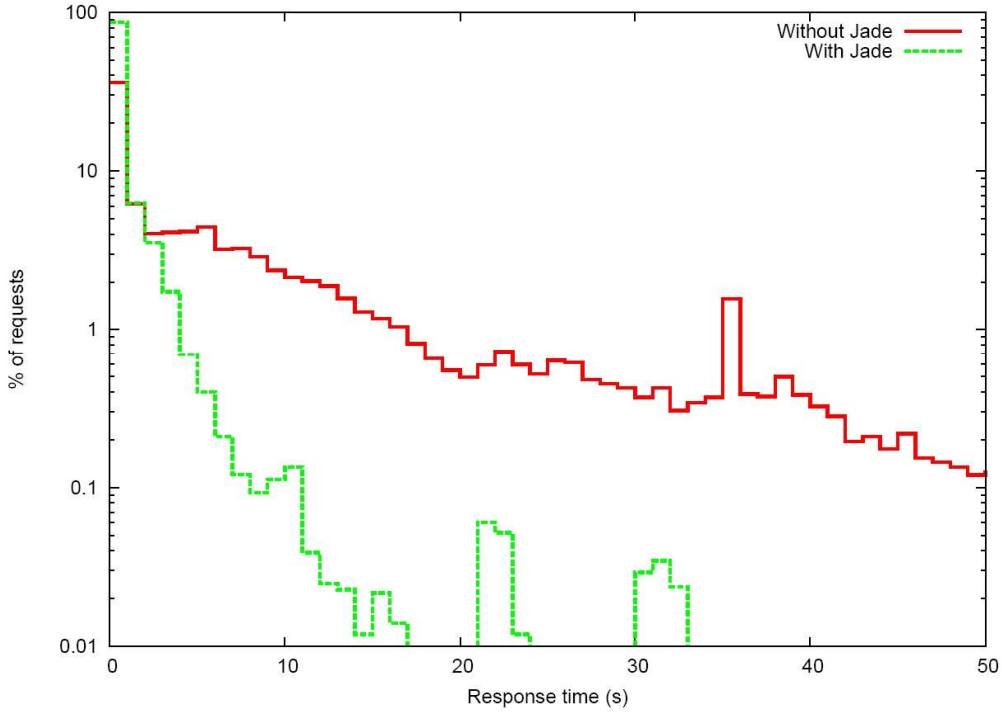

Figure 10. Web client response time variation

As a result of these experiments, when the auction site is managed by Selfware, the number of processed requests is more than twice higher than the number of processed requests by the same system with no self-management features, i.e. 92512 with Selfware vs. 41585 without Selfware. Furthermore, Self-Optimization of the auction site has a clear impact on web client request response times. Figure 10 is a histogram that details the distribution of the response times among all the requests that were issued during the experiments with and without Selfware. This figure shows that the proportion of the requests that are returned within one second reaches 86.25% when

Selfware is used, whereas without Selfware, 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 Selfware, and 48.28% without Selfware. Finally, while with Selfware, no requests were returned in more than 40 seconds, still 5.5% of the requests processed by the non-self-managed system are returned in more than 40 seconds.

These first experiments on Self-Optimization only show the behavior of the auction site in case of gradual variations of the workload, and not in case of load peaks which may also happen. Indeed, in case of a load peak, performing successive reconfigurations to add/remove a single replica might result in a long global reconfiguration before the system stabilizes. Whereas in presence of a load peak, a unique reconfiguration operation that adds/removes several replicas at once should be more effective. We thus conducted additional experiments that exhibit the ability of Selfware to deal with load peaks. Here, the Self-Optimization manager uses an aggregation of sensors that monitor the CPU usage and the number of processed requests on replicated servers. These sensors are used to predict, both the time at which a cluster of replicated elements must be resized, and the number of necessary replicas to add/remove at once. Figure 11 gives the web client request response times variation in presence of a load peak (+1000 clients). Initially, the multi-tier auction site consists of a single enterprise server and a single database server. When the load peak occurs, this has a direct impact on the CPU usage of the database tier which becomes a bottleneck. Thus, Selfware automatically provisions additional database replicas to face the load increase and keep the application performance stable, forming a system consisting of one enterprise server and three replicated database servers. Here, the number of additional replicas to provision is calculated as a function of the number of concurrent processed requests on the bottleneck server. Whereas without Selfware, the application performance simply gets worse, resulting in a trashing.

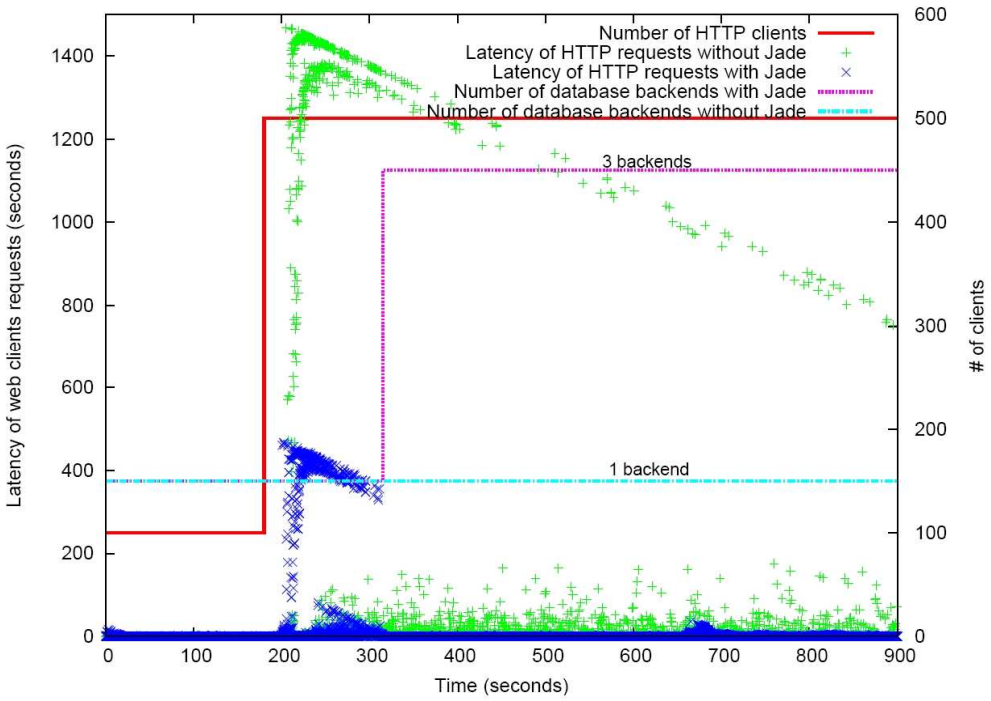

Figure 11. **Behavior of the database tier in presence of load peaks**

3.3.5 Performance overhead

In order to measure the possible performance overhead induced by the self-management framework, we compared two executions of the same multi-tier system: when it is run over Selfware and when it is run without Selfware. During the experiments, the managed application has been submitted to a medium workload so that its execution under the control of Selfware induced no dynamic reconfiguration. The results show no significant overhead in terms of application response times and throughput. We can notice a slight memory overhead (20.1% vs. 17.5%) that can be linked with the creation of internal software components by Selfware. However, Selfware does not induce a perceptible overhead on CPU usage; this is due to the fact that Selfware does not intercept application communications but only configuration/management operations.

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

Figure 12. Performance Overhead

3.4 Self-repair of J2EE 1.4 Application Server with integrity constraints

This scenario is used to repair a transient failure in a JOnAS instance. A transient failure is for instance a memory leak or a memory overload, and a common way to repair such a failure is to restart the application server. Moreover we demonstrate in this scenario some abilities of the dynamic reconfigurations used for system adaptation in the Selfware platform. Actually, it is possible to add some integrity constraints on the cluster architecture that must not be violated during reconfigurations. Then the reconfiguration service ensures that the reconfigurations used by the autonomic manager to repair the system are reliable and conforms to the constraints.

For this scenario, we use the France Telecom web application SOAPSOO described in the next section and we deploy it on a JOnAS 4.8 cluster.

3.4.1 Additional Services used in the scenario

To implement the scenario, we use two additional services in the Selfware platform: the Constraint Checking Service and the Reconfiguration Service.

The ***Constraint Checking Service*** allows to check that the system architecture conforms to some integrity constraints. An integrity constraint is a predicate on assemblies of architectural elements and component state (cf. section 2.3.2 of SP2-L1). Therefore, a system is consistent if all integrity constraints in the system are satisfied. Constraints are expressed with a navigation and selection language used to introspect Fractal system at runtime called FPath. In the Selfware platform, constraints can be specified both on the global architecture of the cluster and on given nodes in the cluster. Some examples of such constraints are the followings:

- On the global cluster architecture:
    - Uniqueness of a JOnAS instance name in the domain
    - Uniqueness of a master instance in the domain to manage the cluster
    - Separation between Web and EJB tiers on different nodes

- On a local node:
  - System resource availability (memory, CPU)
  - Uniqueness of port between JOnAS instances
  - Restricted number of JOnAS instances on the same node

The ***Reconfiguration Service*** is used to dynamically reconfigure the system with a DSL for reconfiguration called FScript. This interpreter is combined to a transactional monitor to ensure ACID properties of dynamic reconfigurations. Thus if a reconfiguration violates some constraints in the target system, the reconfiguration can be cancelled, which means that the transaction is rollbacked and the system is put in its last consistent architectural state before the execution of the reconfiguration by undoing all reconfiguration operations.

3.4.2 Scenario implementation

We choose to repair a memory overload in a JOnAS instance which can leads to a JVM crash by rebooting the server (i.e. the JVM) locally. We put a local constraint on each node of the cluster related to the minimum quantity of available memory which is needed when restarting a server on the node.

The main autonomic control loop (cf. Figure 13) used to repair the transient failure is composed of the following elements:

- Sensors: the MBeanCmd tool provided by JASMINe is used to monitor JOnAS instances in the cluster and allows to catch OutOfMemoryException thrown by failed JVMs.
- Controller: the Reboot Manager subscribes to JMX events thrown by MBeanCmd and when it is notified of a memory overload, it decides to reboot the failed application server.
- Actuators: the ***Reconfiguration Service*** reboots the failed JOnAS instance on the same machine with respect to the integrity constraints. It kills the application server and restarts it. The ***Constraint Checking Service*** checks that the available memory on the node is above the threshold given by the node constraint, if not the ***Reconfiguration Service*** cancels the repair action and notifies the controller and the administrator of the constraint violation.

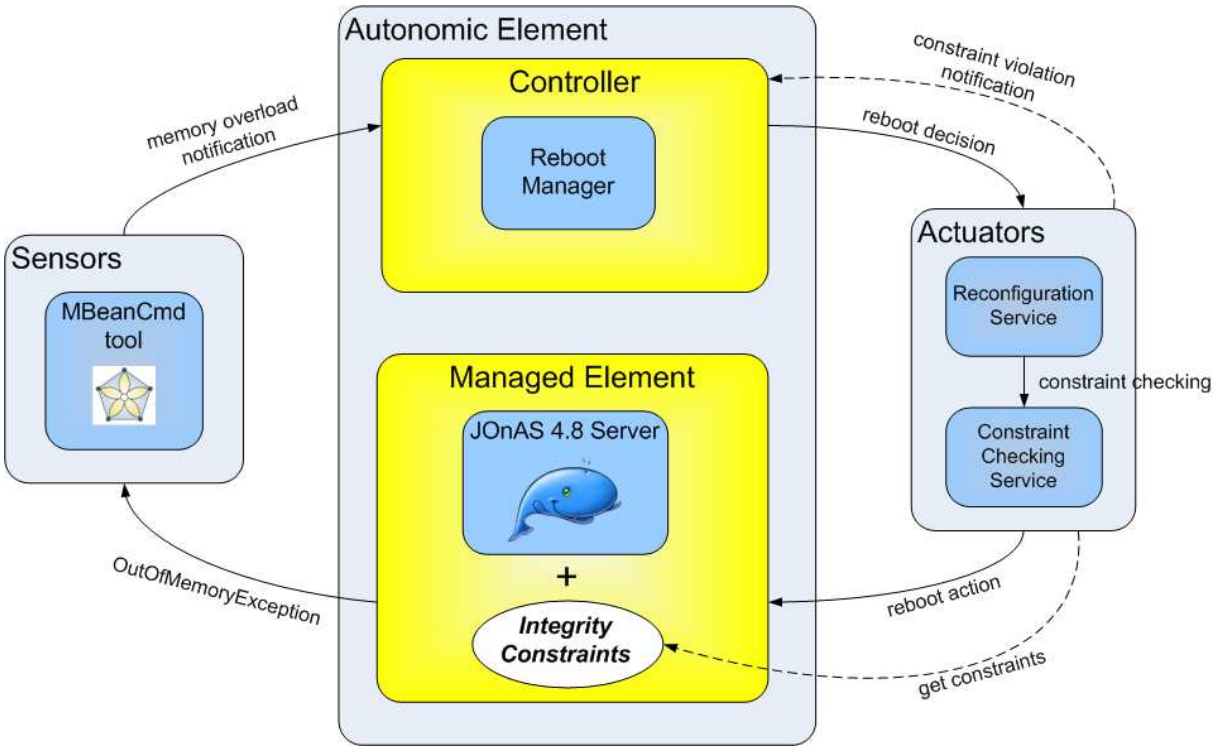

Figure 13.Architecture of the Self-repair management with constraints

A possible extension of this scenario is to react to the constraint violation by building a new repair plan. For instance, the controller can try to reinstanciate the failed server on another node where more memory is available.

3.5 Self-optimization of Java EE 5 Application Server

On this platform we mainly focus on self-benchmarking of Java EE application server and self-optimization of load-balancers. The reason to change of environment was motivated by the recent availability of new versions of load-balancers (at web and ejb levels), much more dynamic. JOnAS 5, by supporting these new load-balancers, eases their optimization by the autonomic manager.

3.5.1 Introduction

In a multi-tiers application, two levels of configuration can be considered to enhance its performance (i.e. throughput-latency ratio) [34]:

1. Architectural configuration by adding/removing replicas

By replicating application servers, the load can be balanced on each replica. So by adding replicas, more clients can be satisfied and the throughput will be increased. If each level is independent and its replication can be managed separately, it is important to notice that the load can spread to other tiers. So if a bottleneck exists at the back-end of the architecture (e.g. a database), this approach can congest the entire cluster and decrease the latency. Figure 14 illustrates such not scalable architectural configuration. The queue symbolizes the pool of connections and the grey squares represent the busy connections. In a bottleneck, no more free connections are available, and many incoming connection requests are waiting.

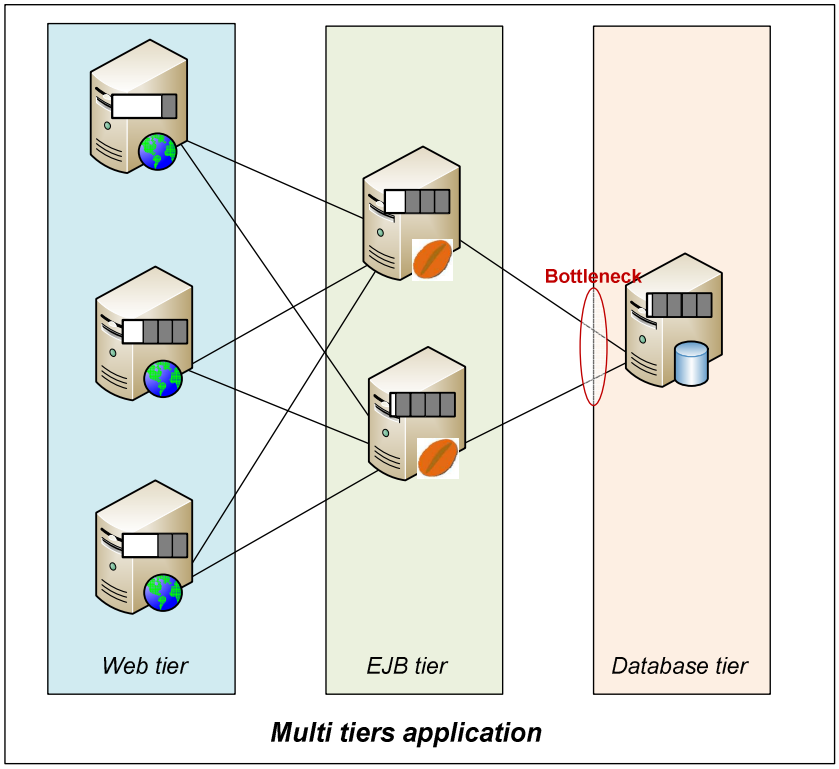

Figure 14. Bottleneck introduced by a bad architectural configuration

2. Local configuration by tuning a replica

By increasing the multi-programming level (MPL) of a local node, this one can accept requests of more clients and increase the throughput. Unfortunately, making pseudo-parallelism introduced a cost due to context switches and can flood the CPU to finally increase the latency. Figure 15 gives the performance result according to the MPL setting. The global performance of the database tier can be, for example, improved by setting the optimal MPL (edge of parabola).

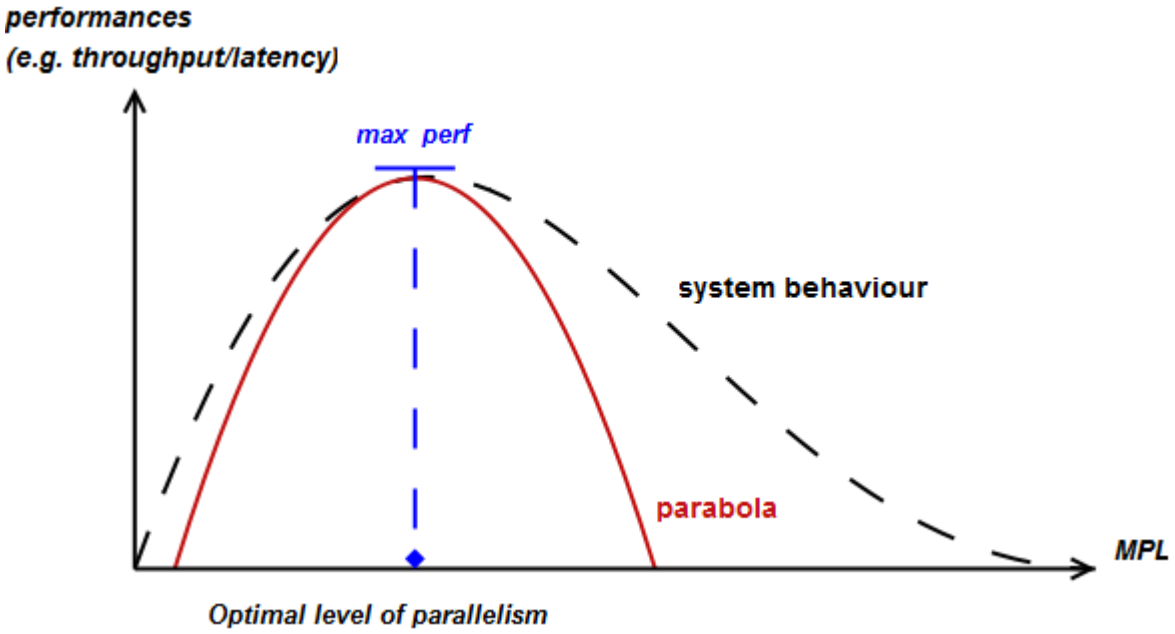

Figure 15. Influence of parallelism on performances

Self-benchmarking can help us to determine the best configurations to get the best performances.

This scenario aims at proposing an auto-optimization in two steps:

- At first, the best MPL configuration is determined through self-benchmarking
- At runtime, the load-balancer parameters are adjusted according to the current load and the optimal performance indicators identified during the first step. Indeed, by knowing capabilities of replicates, we can improve the use of free resources.

3.5.2 Experimental environment

3.5.2.1 The managed Java EE application server JOnAS 5

The SOAPSOO application (cf. section 3.2.2) is deployed on JOnAS 5, the last version of the Java EE application server from OW2. Completely rewritten to be based on OSGi™ modules, JOnAS 5 implements a service-oriented architecture in the application server itself, enabling the server and its services to be dynamically adapted and extended depending on users' needs and the constraints of their environment. The resulting architecture is as in the Figure 16.

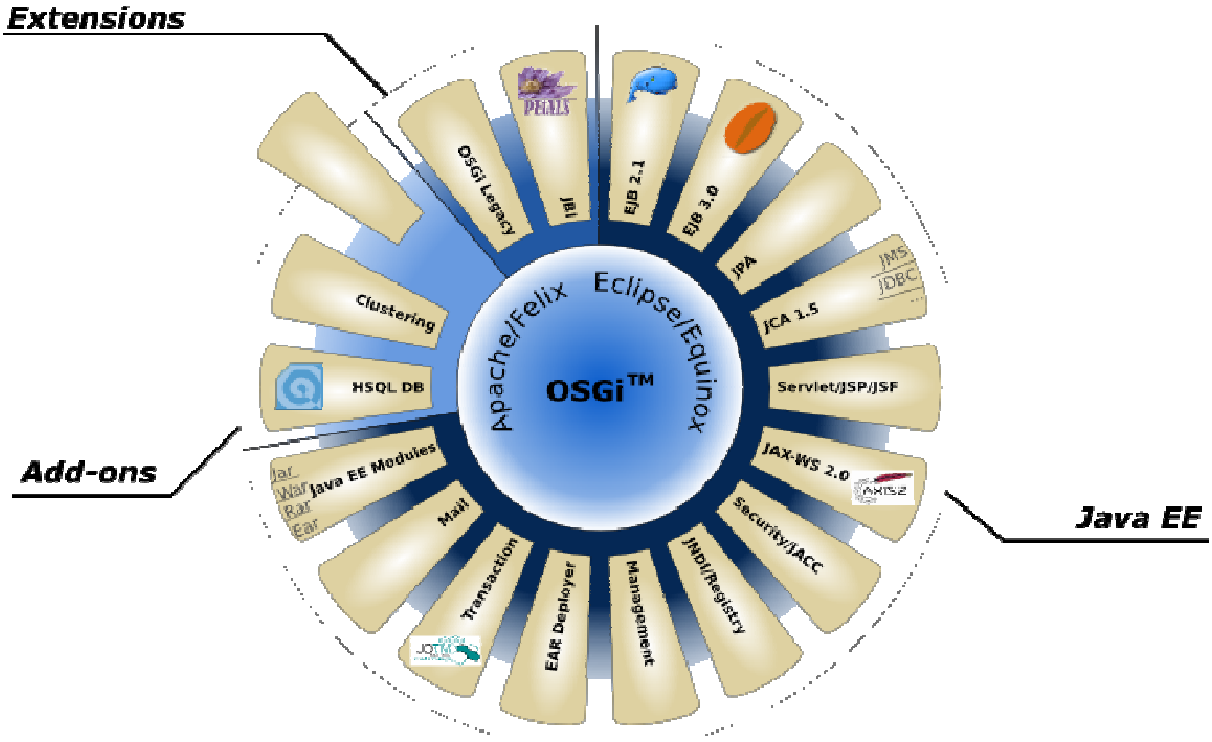

Figure 16. JOnAS 5 architecture

The main benefits of such architecture for an application server are:

- Dynamic configuration and re-configuration of servers
  - Services can be stopped, reconfigured, and started at runtime
  - On demand incremental services delivery: services can be started when required by other services or applications
- Modularity
  - Services are delivered in « bundles »
  - Code readability
  - Reduced system footprint by starting only the strictly required services and by stopping the no more required ones
  - Explicit dependencies are clearly defined between services

- OSGi other add-ons
  - Remote management
  - Services lifecycle management (incl. versions)
  - Services dependencies management (OBR: OSGi Bundles Repository)
  - Dynamic class loading (flat class loaders)
  - OSGi world accessibility (RFID, probes …)

3.5.2.2 Wildcat

Wildcat is a generic framework for context aware applications [36]. It provides a simple and dynamic data-model to represent the execution context of the application, and offers a simple API for the programmers to access this information both synchronously and asynchronously (*pull* and *push*). Wildcat allows to scale numerous data-sources and events:

- Firstly, it hierarchically organizes data-sources (as depicted on figure 20): each data-source is attached to a leaf in a tree structure. Operations on hierarchy are similar to file system usages: data sources are mount/unmount, data sources are denoted by their path in the hierarchy…

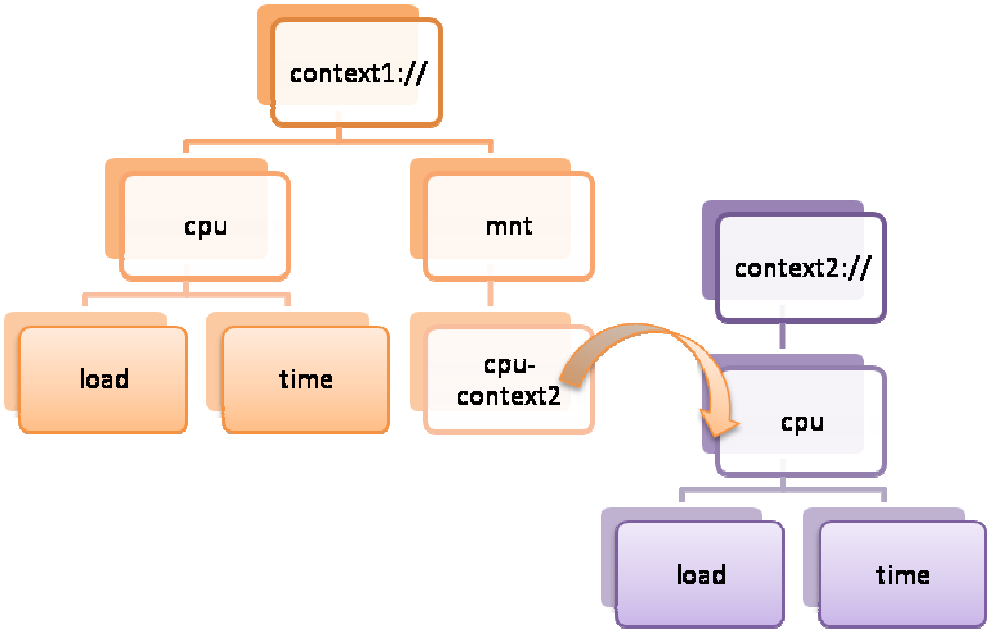

Figure 17.Hierarchical classification of event producers

- Secondly it embeds a CEP engine (for Complex Event Processing [35]) that can process large volumes of events.

It features a query language that allows to specify sliding windows to group, aggregate, sort, filter and merge event streams. Thus one can easily express synthetic data and event patterns upon which to trigger a reconfiguration.

Instances of Wildcat can communicate through RMI for pulling events, and through JMS for pushing them. Hence, one can easily examine execution contexts of distributed applications.

3.5.2.3 Self-benchmarking with Clif

Here, we try to get optimal JOnAS configuration parameters for the SOAPSOO web application through an autonomic benchmarking campaign, before actually deploying the application. As a matter of fact, although Selfware is aiming at providing self-optimizing systems once deployed, deploying a

badly configured web application, hoping it will quickly and accurately self-optimize is certainly not a good idea for quality of user experience reasons, or even for the mere stability of the application. Thus, the idea is to deploy a pre-optimized application, and more precisely a self-optimized application, thanks to the Selfware-based Autobench architecture, as described in [4].

So, we apply to the SOAPSOO web application the Autobench architecture, strategies and algorithm described in SP2-L2 for self-benchmarking (see Figure 18). As presented in section 3.5.1 and Figure 15, we will try to self-optimize the maximum number of threads used by JOnAS to process the incoming request (MPL parameter).

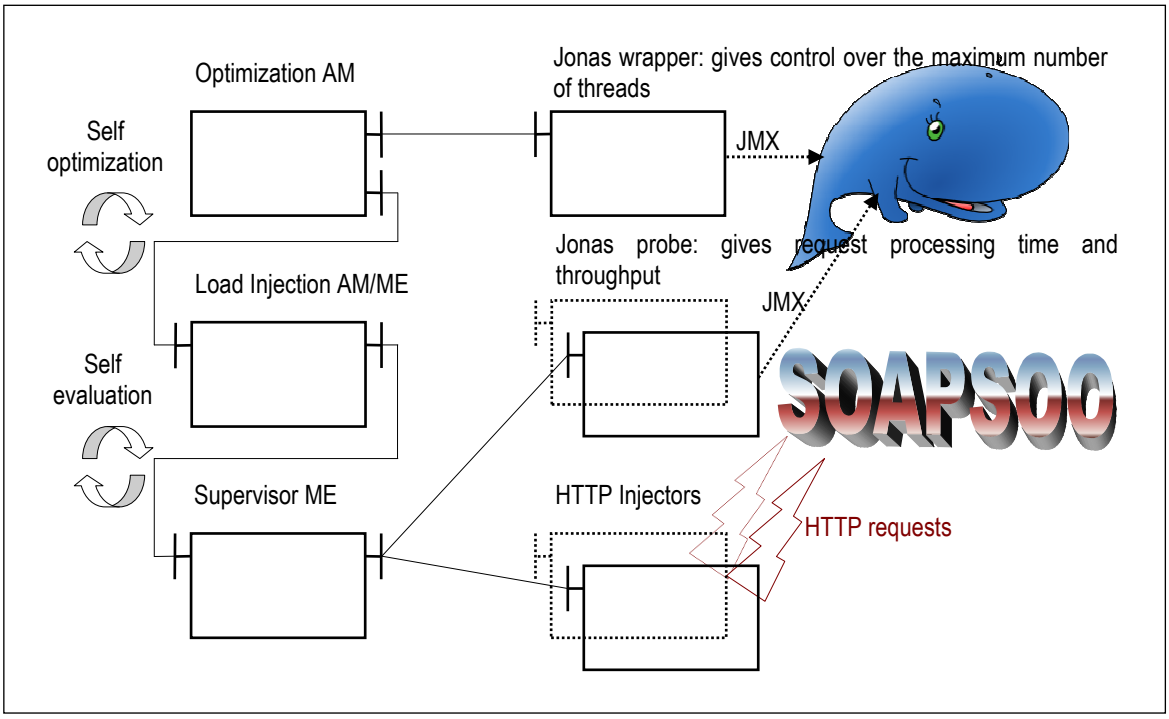

Figure 18. The Autobench architecture applied to the SOAPSOO application over Jonas

To do this, we require these specific elements:

- a JOnAS wrapper that enables setting the MPL. This wrapper uses JMX to change JOnAS' maxThreads parameter. A stop()/start() cycle must be applied to JOnAS' HTTP connector (also via JMX) in order to take into account the new setting;
- a CLIF probe for JOnAS, giving load metrics to the Load Injection AM, in order to regulate the autonomous load injection. This probe gets the average request processing time and throughput from JOnAS via JMX[3];
- a CLIF Http Injector to generate HTTP request on SOAPSOO. This is already available in the ISAC scenario module for CLIF;
- a definition of a virtual user behavior for CLIF's ISAC scenario module. This behavior defines typical user requests and think times for the SOAPSOO application. The Load Injection AM will autonomously adapt the number of active virtual users according to the observed load.

[3] An alternative is to rely on the load injectors which deliver similar metrics but also complementary metrics, from the client perspective (i.e. including the network latency). For instance, load injectors give the minimum, maximum, standard deviation and average value for the response times. But these measures are likely to be polluted by garbage collector occurrences in the load injector JVM. So, we have to experiment both ways: more metrics or more accuracy.

The Optimization AM generates possible values for the maxThreads parameter, trying to maximize the throughput/latency metric. This metric must be computed and delivered by the Load Injection AM.

3.5.2.4 The managed load-balancers Apache/JK and CMI

JK load-balancer element is plugged in the Apache element. It is localized at server side (Figure 19).

CMI load-balancer element is embedded in the EJB API and this localized at the client side (Figure 20).

Load-balancer of web accesses (Apache/JK)

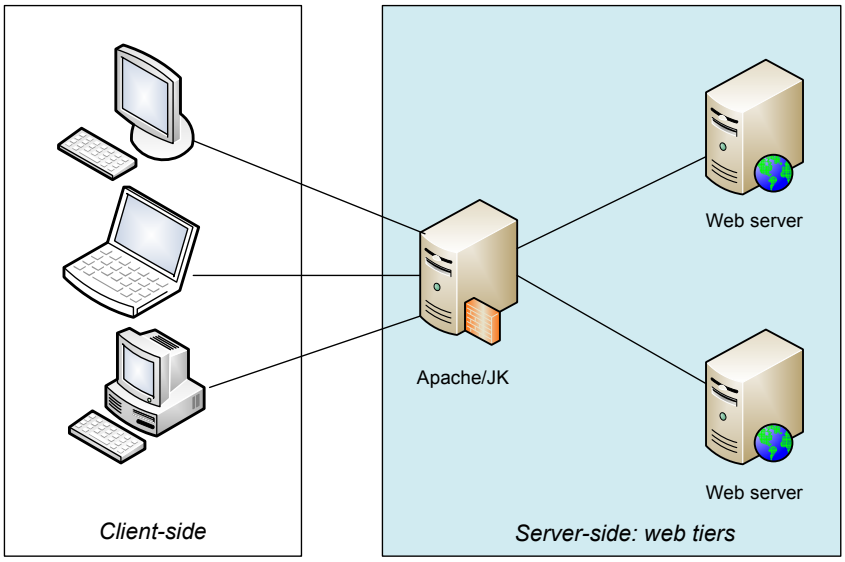

Figure 19. Load-balancer of web accesses

The Selfware system provides a self-optimization manager supporting some dynamic optimizations of JK plug-in web load-balancer. The manager is able to perform a self adjustment of the load-balancing factors (weights) for each ME. Sensors get different indicators for each ME: memory, CPU, bandwidth, and many other application server metrics, such as pool size, thread number, etc. The autonomic manager collects these monitoring data and aggregates the load indicators (e.g. throughput/latency). Finally, when a load indicator raises a given limit, the autonomic manager determines the new load-balancer factors for each ME and set them through the actuator.

Load-balancer of EJB accesses (CMI)

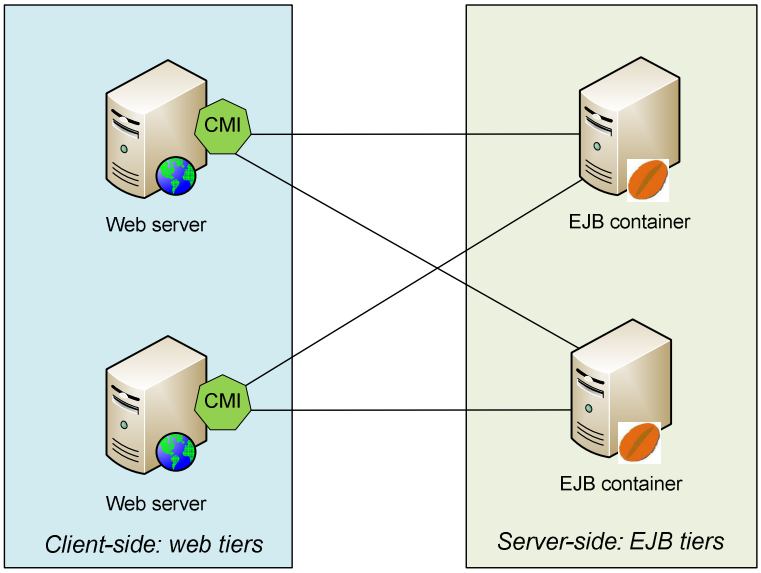

Figure 20. Load-balancer of EJB accesses

The Selfware system provides a self-optimization manager supporting some dynamic optimizations of the EJB/CMI load-balancer. Its capabilities are:

- Self optimization of load-balancer factors: just as JK, the manager is able to determine and set the best load-balancing factors for ME according the current load of each ME.
- Self evicting: when the load of a ME becomes critical, the autonomic manager is able to disable temporary this one in order to refuse connections. Existing connections are still served. When the load of an evicting ME decreases under a threshold, this ME is again enabled.

The next section describes the implementation of this self-optimization manager.

3.5.2.5 Implementation details of the self-optimization manager for load-balancers

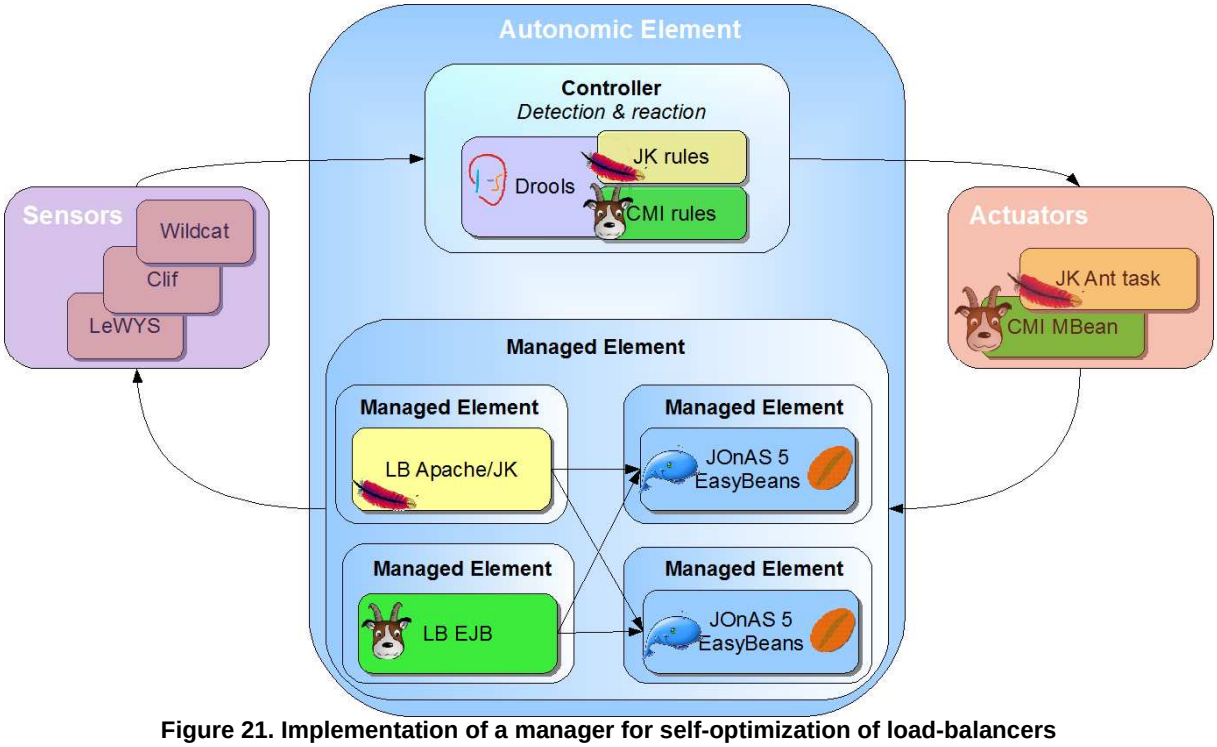

Figure 21. Implementation of a manager for self-optimization of load-balancers

Figure 21 describes the main parts of the manager:

- *Sensors*

Measures are collected by using Wildcat. This one acts as a higher layer of sensor which is in charge of performing a first treatment of measures. Measures are extracted from the following sources:

  - Clif and LeWYS probes (e.g. CPU, network, memory) whatever the operating system.
  - JMX probes, exposed by the application server, provide some useful indicators (e.g. pool size, thread number). The MPL can be computed from these (from the throughput-latency ratio).

- *Controller*

It is implemented by using the rules engine Drools. This one allows dynamically adding or removing bundles defining rules and actuators for given load-balancers.

  - Detection

Since inserted events are facts, the Drools engine uses events as input. Detection is performed, on these events, by using the LHS of rules. LHS (for left-hand side) is the conditional parts of a rule.

  - Reaction

The RHS (for right-hand side) of rules implement reactions and invoke actuators. RHS is basically a block that allows dialect specific semantic code to be executed.

- Actuators

Reconfiguration of legacies is performed by using their wrapper components. According to the load-balancers, these wrappers uses:

  - A MBean to reconfigure CMI
  - An Ant task to reconfigure JK.

3.5.2.6 Scalability of the autonomic manager

The autonomic manager is designed to be scalable by combining an event-driven architecture (distributed) with a business rule management system (cf. Figure 22):

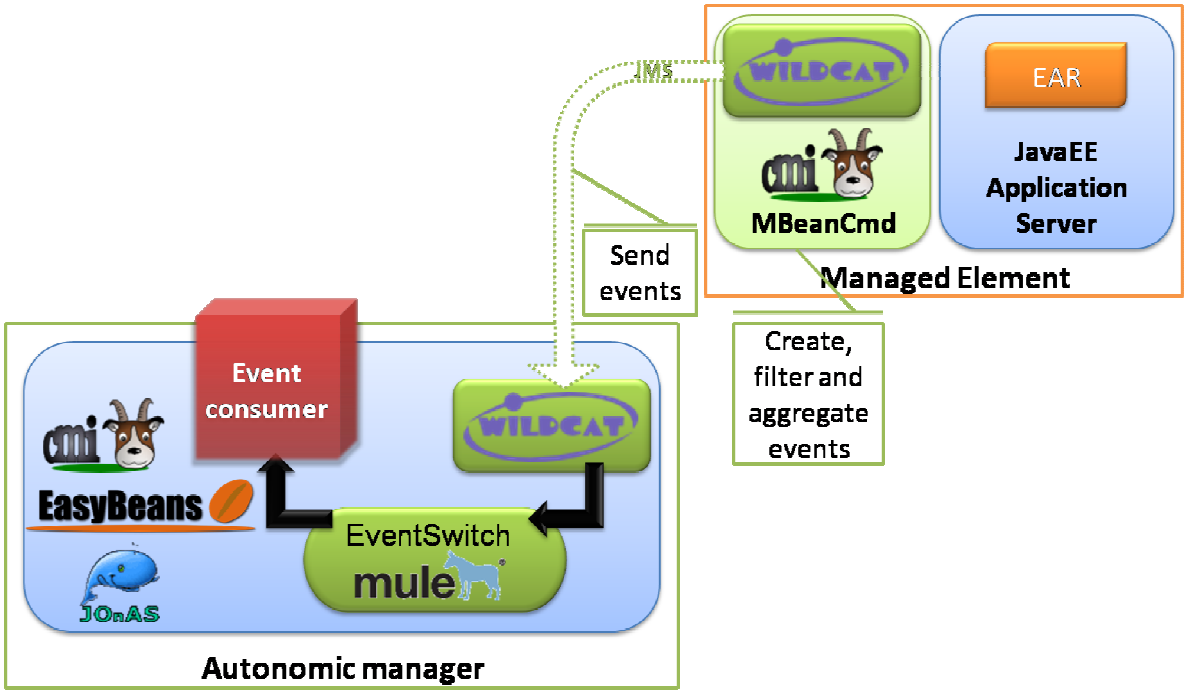

Figure 22. An event-driven architecture

WildCAT assumes the role of event generator and event channel. Routing and mediation is performed by EventSwitch. Complex event processing is supported by WildCAT. Finally a business rule management can be one kind of event consumer (cf. Figure 23).

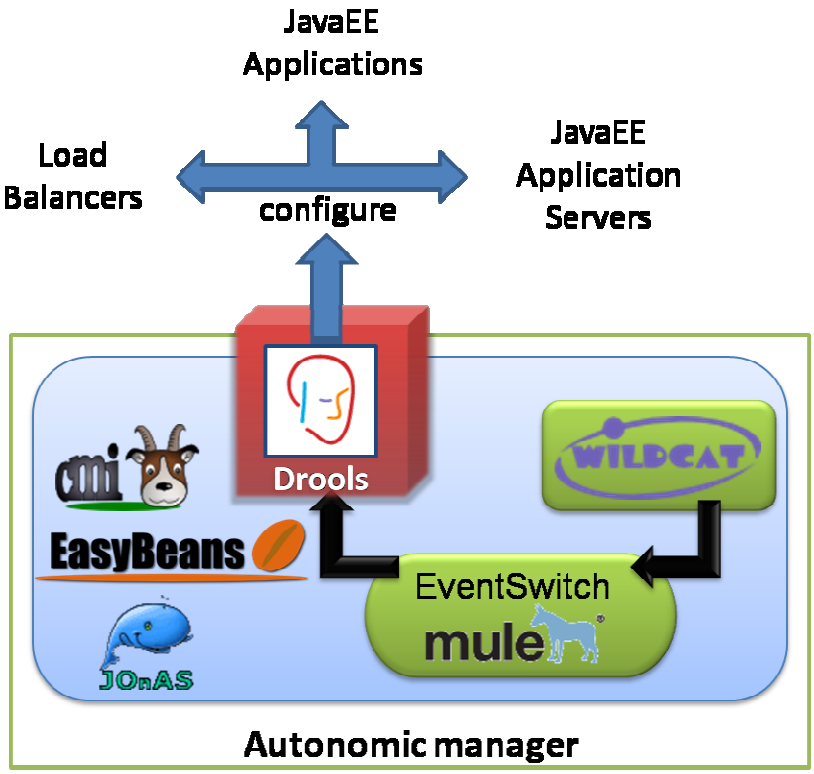

Figure 23.Using a BRMS as event consumer

3.5.2.7 Deployment architecture

The following figure gives an example of deployment for self-managing a cluster of two JOnAS nodes:

- One machine for hosting the load-balancer (managed element)
- Two machines for hosting the J2EE server nodes (managed element)
- One mchine for hosting the autonomic manager

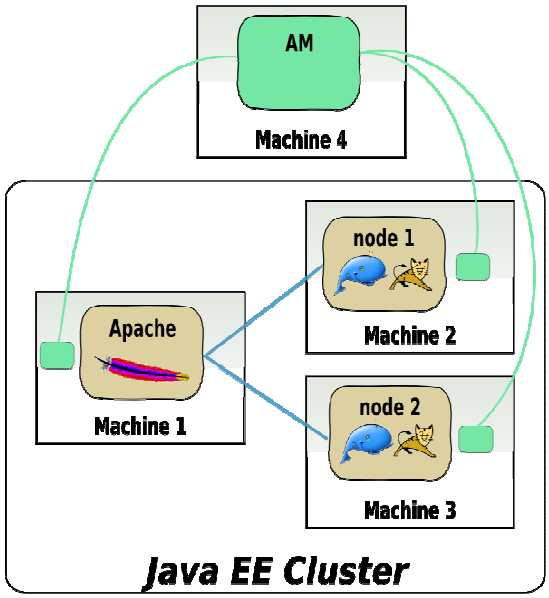

Figure 24.Deployment architecture

3.5.3 Approach for self-optimization

The scenario can be divided in two successive steps. Firstly, an optimal configuration of MPL is computed by self-benchmarking. Secondly, load-balancers are dynamically tuned in order to respect the performance of each replicate.

3.5.3.1 Looking for an optimal configuration by self-benchmarking

As explained in section 3.5.2.3, our goal is to get the Autobench architecture autonomously find the best setting for JOnAS maxThreads parameter (MPL), i.e. the maximum number or parallel request executions in JOnAS. We define the best setting as the value that results in the maximum throughput/latency metric.

Because of the specificity of this parameter, a dichotomic search based on an a priori minimum value and maximum value is not directly applicable. First, there is no intrinsic maximum possible value; second, even though we could think of a few thousands threads as a practical limit, it could be dangerous for the system stability to start with so many threads, or even half of them. Remember that we want to avoid any crash of the System Under Test in order (1) to implement simpler Load Injection and Optimization AMs and (2) to go quicker in the self-benchmarking campaign[4].

Then, we can think of an algorithm that roughly gives the necessary bounds, before actually applying the dichotomic search. The lower bound is not really a problem, since we can at least choose one (single-threaded server). The upper bound needs more attention, because we have to find a maximum MPL limit that is actually greater than the optimal MPL value, but also not too far from the optimal value to avoid instability problems. Moreover, we want to be more efficient (quick) than successively trying all possible values, incrementing the MPL step by step from one until the performance metric (throughput/latency) starts to decrease.

[4] We could insert a self-repair manager here, in order to recover from a SUT crash. However, this would add a third AM in the architecture, with an extra control loop. Coping with these loops would require a more complex orchestration. It would also significantly slow the self-benchmarking process since restarting a faulty node takes a significant piece of time. Nevertheless, autonomic management of several, heterogeneous autonomic managers is an interesting topic for future work directions.

To go quick without making any assumption about the order of magnitude of maximum MPL value, a possible principle is to successfully double the MPL value, make an experiment with this new value, and to check if the performance metric is lower than the previous experiment. This is a way to go quick for linear systems (although the algorithm would never end), but we know that our metric will reach a maximum and then decrease, possibly (but not necessarily nor strictly) following a parabolic curve (see Figure 15). In other words, we can assume that the performance metric will increase more and more slowly, and then decrease faster and faster while the MPL grows. In order not to go far beyond the optimal MPL value, and possibly beyond the system operating limit, we must temper the arithmetic progression of the MPL values by a tempering factor, e.g. based on the evolution of the curve slope between two consecutive points.

More formally, let m_i be the measured metric for experiment number i, and let MPL_i be the maxThreads value for this experiment. The curve slope S_{i+1} from point (MPL_i, m_i) to point (MPL_{i+1}, m_{i+1}) equals (m_{i+1}-mi)/(MPL_{i+1}- MPL_i). To temper the arithmetic progression, the MPL increment is the latest MPL value multiplied by the tempering factor (S_{i+1}/S_i): $MPL_{i+2} = MPL_{i+1} \times (1 + (S_{i+1}/S_i))$. For linear systems, this factor is constant and equals to one, so we keep doubling the MPL values from one experiment to the following. For the kind of system we are testing, this factor may be very close to one during the first experiments (low MPL values), and then progressively decrease, playing the wanted tempering role. Assuming function f / $m_i=f(MPL_i)$ is close to a parabolic function, we can make the tempering factor more efficient by using its square value. Finally, we get the algorithm given by Figure 25 and illustrated by Figure 26.

```
// let mi be the performance metric from experiment number i
// let MPLi be JOnAS' maxThreads value for experiment number i
// the MPL values for the first three iterations are manually set
// because the tempering factor is undefined
MPL1 = 1
MPL2 = 2
MPL3 = 4
m1 = experimental_evaluation(MPL1)
m2 = experimental_evaluation(MPL2)
S2 = (m2- m1)/(MPL2- MPL1)
m3 = experimental_evaluation(MPL3)
S3 = (m3- m2)/(MPL3- MPL2)
i = 3
while mi > mi-1
      MPLi+1 = MPLi x (1 + (Si/Si-1)^2) // tempered arithmetic increment
      mi+1 = experimental_evaluation(MPLi+1)
      Si+1 = (mi+1- mi)/(MPLi+1- MPLi)
      i = i + 1
// the maximum bound is MPLi
// finally, we also get MPLi-2 as a good minimum bound
// then, we can perform a dichotomic search within this range
```

Figure 25. Tempered arithmetic search algorithm for the MPL bounds

As a side result, if max is the index of the MPL maximum bound found by the algorithm, we also get MPL_{max-2} as a good minimum bound (MPL_{min}), in the sense that it is closer to the optimal MPL value than all other values MPL_i with $0<i<$ max-2. Note that nothing can be said about MPL_{max-1}. These results are valid for any function whose derivative is monotonic.

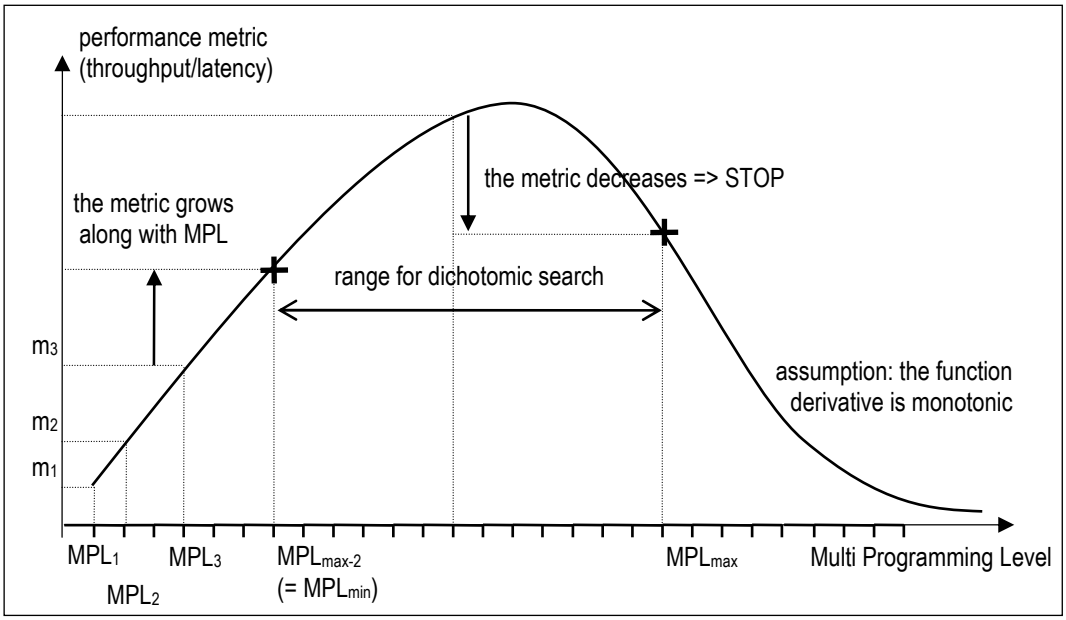

Figure 26. Illustration of the bounds search algorithm

3.5.3.2 Dynamic reconfiguration of the load-balancing algorithm based on the current activity of cluster nodes

To launch a reconfiguration, the optimization manager needs to detect some abnormal performances of replicated MEs. By breaking performances of a particular replicated ME, we can cause a dynamic reconfiguration of the load-balancing algorithm in order to lighten the request throughput toward this. Three reasons (non-exclusive) can be considered for differences of performance between replicated MEs:

- Replicated MEs are heterogeneous.

This case correspond to a local weakness of a replicated ME. Heterogeneity appears in the following cases:

- When hardware are different.
- When a replicated ME is not dedicated to the application (for example the same machine hosts some batch applications)
- When the configuration is not optimal.

An example (Figure 27) is when a replicated ME accept few connection requests (e.g. due to a not efficient configuration). Its latency is of course minimal, but its MPL setting is needlessly low. Consequently its request throughput is limited that breaks the performances of this replicated ME.

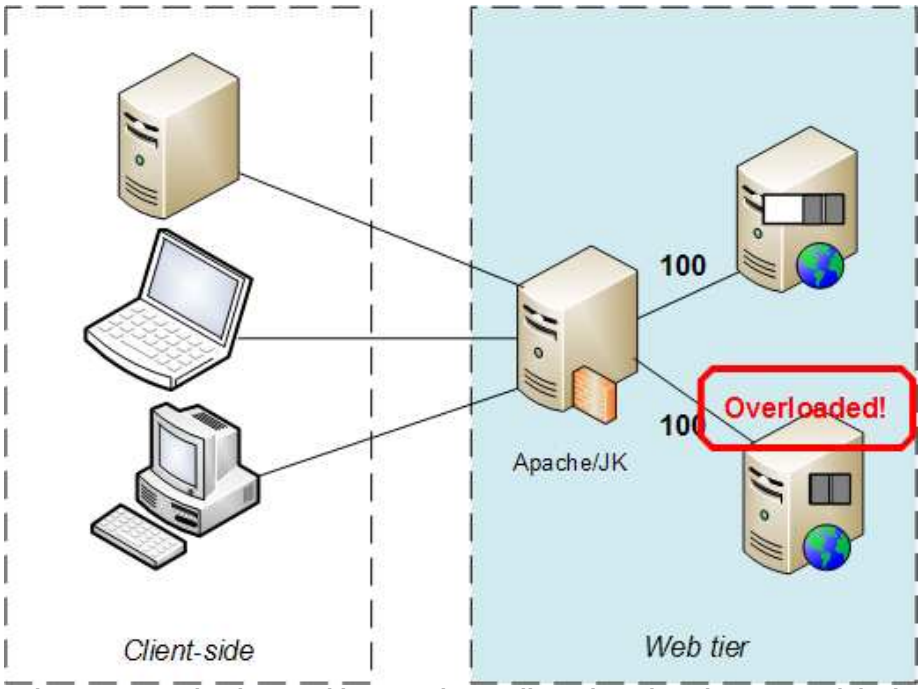

Figure 27. Overload caused by a weaker replicated ME (numbers are weights)

- Load-balancing is not fair, i.e. the incoming requests are not equitably distributed.

The performances of a replicated ME can be broken by an imbalance (e.g. by adding some clients that bypasses the load-balancer as in Figure 28).

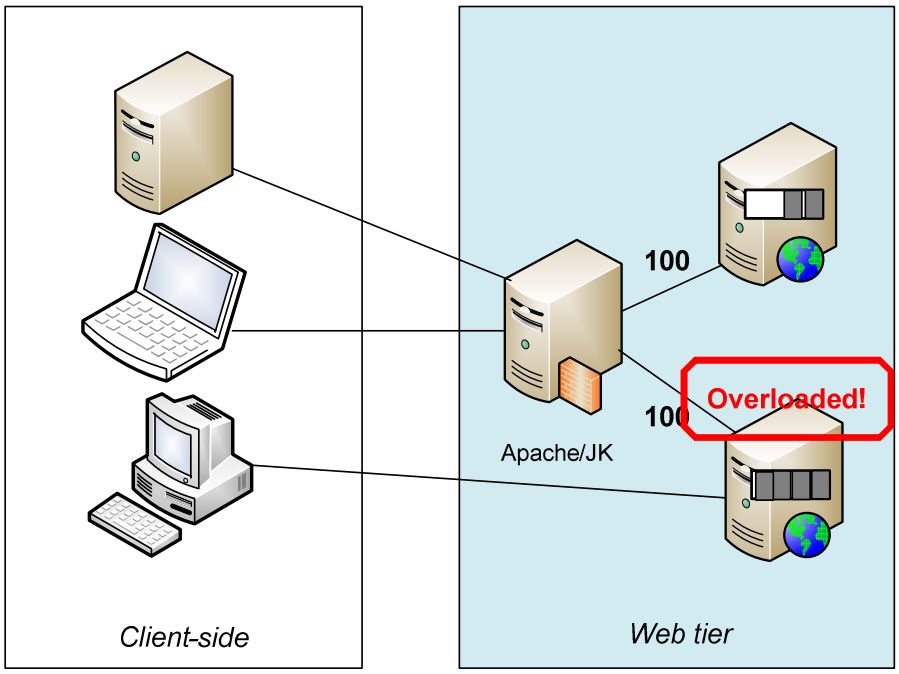

Figure 28. Overload caused by an imbalance of the load distribution

- A replicated ME is linked to another overloaded tiers

The latency can be increased if the replicated ME is linked to an overloaded tier (for example a LDAP as in Figure 29).

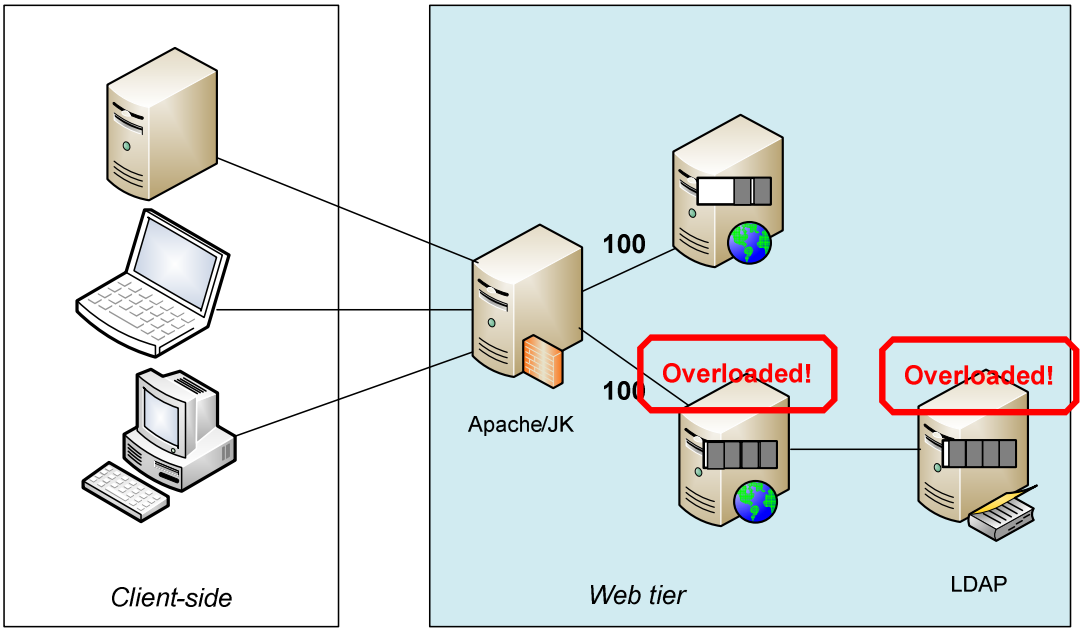

Figure 29. Overload caused by the link to an overloaded LDAP

After having detected an abnormal state (coming from any of three reasons), the autonomic manager will readjust the load-balancer (Figure 30).

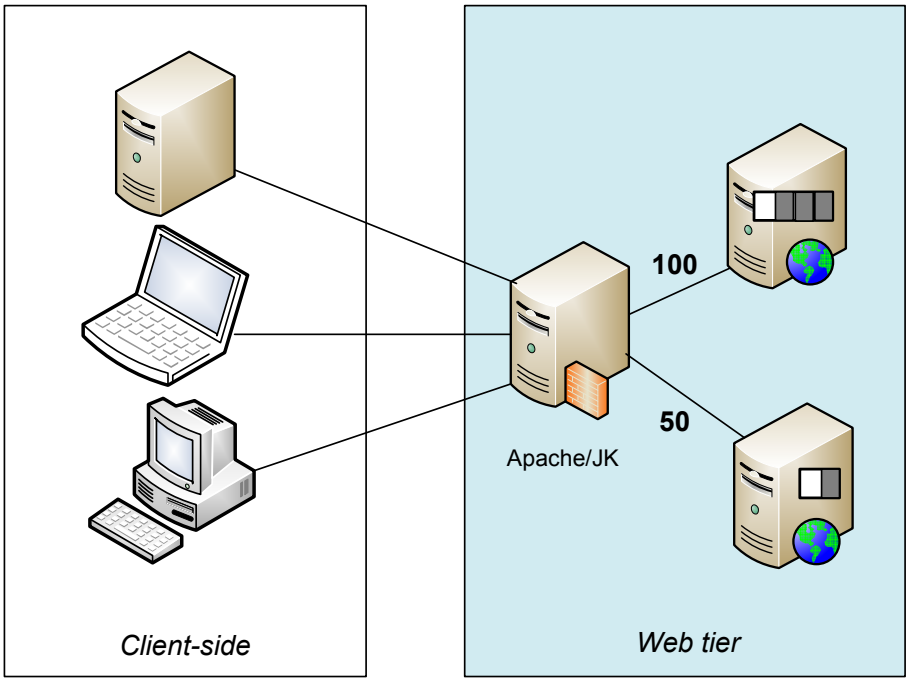

Figure 30. Reconfiguration of the load-balancer due to the heterogeneity of MEs

We dispose of two kinds of indicators in input of the analyser (rules engine), which detects abnormal performances:

- Material resources (CPU, memory, network)
- Software resources (OS, JVM, application server)

Self-benchmarking gives an initial base of optimal indicators for software resources. In this scenario the base won't be updated and the computed optimal MPL, which belongs to the base, is an input of the analyzer (as a reference value).

4 Conclusion

Nowadays J2EE servers are widespread in the information system of the enterprise and more and more critical applications are deployed on top of such middlewares. The J2EE clustering mode provides a solution for scalability and high availability but increases highly the complexity of the administration.

Self-management addresses this problem and provides some autonomic behaviour such as self-healing and self-optimization.

This document has described the three scenarios that were experimented in the Selfware project regarding autonomic management of J2EE application servers. The first scenario presents a set of autonomic managers performing both the self-repair and the self-optimization. The second scenario focuses on an autonomic manager making repairs according to some given integrity constraints. The third scenario focuses on a joint use of self-benchmarking and self-optimization of load-balancers.

These experiments do validate the interests for self-* in the J2EE server management. When combined with the clustering mechanism, self-managed applications have a better quality of service in terms of performance and availability.

The selfware approach, based on an external control loop, is not intrusive and can be applied to third party products. For example, it has permitted to implement easily the third scenario with two different load-balancers Cmi and Jk without any changes in the managed element. Adding another load-balancer would be simple as well.

In Selfware the scenarios are focuses to the J2EE layer and deal with the J2EE server at a coarse grain. Possible further works could address:

- The whole system for managing both the OS virtualization layer and the Java EE layer.

- The J2EE server at a finer grain, new generation of application servers, OSGi based, are becoming highly modular and dynamic what add a degree of complexity in the monitoring and reconfiguration operations.

5 References

[1] Selfware RNTL Project, Livrable SP1-Lot1, sardes.inrialpes.fr/~selfware, August 2007

[2] Selfware RNTL Project, Livrable SP1-Lot2, sardes.inrialpes.fr/~selfware, November 2007

[3] Selfware RNTL Project, Livrable SP2-Lot1, sardes.inrialpes.fr/~selfware, November 2007

[4] Selfware RNTL Project, Livrable SP2-Lot2, sardes.inrialpes.fr/~selfware, June 2008

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

[6] Apache - HTTP Server Project. Apache. http://httpd.apache.org/.

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

[8] 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.

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

[10] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. A Microrebootable Sys- tem: Design, Implementation, and Evaluation. In *6th Symposium on Operating Sys- tems Design and Implementation (OSDI-2004)*, San Francisco, CA, December 2004.

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

[12] 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, Ban , Canada, October 2001.

[13] Y. Chawathe and E. A. Brewer. System Support for Scalable and Fault-Tolerant Internet Services. In *Distributed System Engineering. The British Computer Society*, 1999.

[14] S. W. Chen, A. C. Huang, D. Garlan, B. Schmerl, and P. Steenkiste. An Architecture for Coordinating Multiple Self-Management Systems. In *4th Working IEEE/IFIP Conference on Software Architecture (WICSA-4)*, Oslo, Norway, June 2004.

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

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

[17] X. He and O. Yang. Performance Evaluation of Distributed Web Servers under Com- mercial Workload. In *Embedded Internet Conference 2000*, San Jose, CA, September 2000.

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

[19] M.Y. 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, April 2001.

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

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

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

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

[24] 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.
[25] Y. Saito, B.N. Bershad, and H.M. Levy. Manageability, Availability and Performance in Porcupine: A Highly Scalable, Cluster-Based Mail Service. *ACM Transactions on Computer Systems*, 18(3), August 2000.
[26] 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)*, December 2002.
[27] Sun Microsystems. Java 2 Platform Enterprise Edition (J2EE). http://java.sun.com/j2ee/.
[28] The Apache Software Foundation. Apache Tomcat. http://tomcat.apache.org/.
[29] Java Open Application Server (JOnAS), http://jonas.objectweb.org.
[30] B. Urgaonkar and P. Shenoy. Cataclysm: Handling Extreme Overloads in Internet Services. Technical report, Department of Computer Science, University of Massa- chusetts, November 2004.
[31] B. Urgaonkar and P. Shenoy. Sharc: Managing CPU and network bandwidth in shared clusters. *IEEE Transactions on Parallel and Distributed Systems*, 15(1), 2004.
[32] B. Urgaonkar, P. Shenoy, A. Chandra, and P. Goyal. Dynamic Provisioning of Multi- Tier Internet Applications. In *2nd International Conference on Autonomic Computing (ICAC-2005)*, Seattle, WA, June 2005.
[33] 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, April 2001.
[34] Jean Arnaud, Sara Bouchenak. Gestion de ressources dans les services Internet. *Conférence Française en Systèmes d'Exploitation (CFSE'6)*, Fribourg, Suisse, Février 2008.
[35] David Luckham. The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems. Addison-Wesley Professional, May 2002.
[36] Wildcat: a toolkit for context-aware applications, http://wildcat.ow2.org.