# Delivery Selfware SP3

## Lot 1, August 24, 2007

# Java EE management scenarios

**Authors**

Noel De Palma (INRIA/Sardes)
Bruno Dillenseger (Orange labs)
Benoît Pelletier (Bull SAS)

**Version**

1.0

# Contents

# 1 Introduction

This document aims at describing a set of autonomous management scenarios for the Java EE cluster.

This paper is organized as follows. Section 2 reminds a few definitions on the autonomic system. Section 3 presents the Java EE context and describes the JOnAS application server and its cluster mode. Afterwards, section 4 gives the motivations for applying the control loop in the Java EE field. Section 5 proposes the use cases.

# 2 Java EE context

The cluster Java EE scenarios are implemented through the ObjectWeb's open-source Java EE platform JOnAS.

## 2.1 Java EE standard

The Sun Java EE specification, together with its related specifications ( EJB, JMS,...), defines an architecture and interfaces for developing and deploying distributed Internet Java server applications based on a multi-tier architecture. This specification intends to facilitate and standardize the development, deployment, and assembling of application components; such components will be deployable on Java EE platforms. The resulting applications are typically web-based, transactional, database-oriented, multi-user, secured, scalable, and portable. More precisely, this specification describes two kinds of information:

- The first is the runtime environment, called a Java EE server, which provides the execution environment and the required system services, such as the transaction service, the persistence service, the Java Message Service (JMS), and the security service.
- The second is programmer and user information explaining how an application component should be developed, deployed, and used.

Not only will an application component be independent of the platform and operating system (since it is written in Java), it will also be independent of the Java EE platform.

A typical Java EE application is composed of 1) presentation components, also called "web components" (Servlets and JSPs), which define the application Web interface, and 2) enterprise components, the "Enterprise JavaBeans" (EJB), which define the application business logic and application data. The Java EE server provides containers for hosting web and enterprise components. The container provides the component life-cycle management and interfaces the components with the services provided by the Java EE server. There are two types of containers; the web container handles Servlet and JSP components, while the EJB container handles the Enterprise JavaBeans components. A Java EE server can also provide an environment for deploying Java clients (accessing EJBs); it is called client container.

## 2.2 JOnAS

JOnAS is an open source application server, developed within the OW2 consortium.

JOnAS is an enterprise class Java EE application server:

- Benefiting from a mature code and the J2EE 1.4 Sun certification. Consequently JOnAS is largely deployed and is robust.
- Providing some high level scalability and high availability mechanisms through a set of tuning parameters at different levels (threads pools, data or connection caches) and though end to end and powerful clustering features
- Equipped with some advanced administration tools such as the jonasAdmin JMX console, the domains management or the shell commands for the production environment
- Facilitating the Enterprise Information System integration through the JCA connectors, the Web services or the LDAP access.
- Offering some development tools through a set of IDE plugins

More detailed informationn about JOnAS are given in appendix.

# 3 Rationale

Java EE clusters are more and more frequently deployed within Enterprise Information Systems, to cope with the growing requirements in term of scalability and availability. In many cases the sudden expansion of such systems has led to very complex structures which, using conventional methods can be only monitored and controlled with difficulty. These distributed architectures raise new administration issues, since they cannot be handled reasonably by hand to satisfy the required QoS:

- The N-tiers model (web/business/database tiers), the stack of several pieces of software (OS/JVM/Java EE/application) and the distributed aspects of a cluster increase roughly the number of indicators to observe for monitoring the system.
- The configuration and tuning capabilities provided by the middleware components, together with the complexity of the Java EE do require the setting of a huge number of parameters.

A system tool is required for simplifying the operator job. It relies on advanced management features and on autonomous behaviour capabilities to reduce the management costs of such architectures. The main benefits are:

- to reduce the risk of human error in configuration and management operations
- to improve the response time to eliminate malfunctions occurring in the system
- to improve the global availability of applications (by minimizing service interruption periods)
- to optimize global performance

The self-management is still a challenge. For instance in a multi-tiers architecture, self-recovery upon an application server failure does not simply imply restarting the failing server as an independent element, but also rebuilding the connections of that server with its front-end (eg. Apache server) and back-end servers if any( eg. Database). This requires the knowledge of the global architecture of the system in order to perform reconfigurations of several elements of the system; and such an architectural view is not provided by distributed systems

# 4 Scenarios

## 4.1 Self-management

### 4.1.1 Cluster architecture

The scenarios are implemented with the following architecture:

- 1 apache server
- 2 JOnAS application server node (node1-2) corresponding to the Java EE middleware
- 1 MySQL database

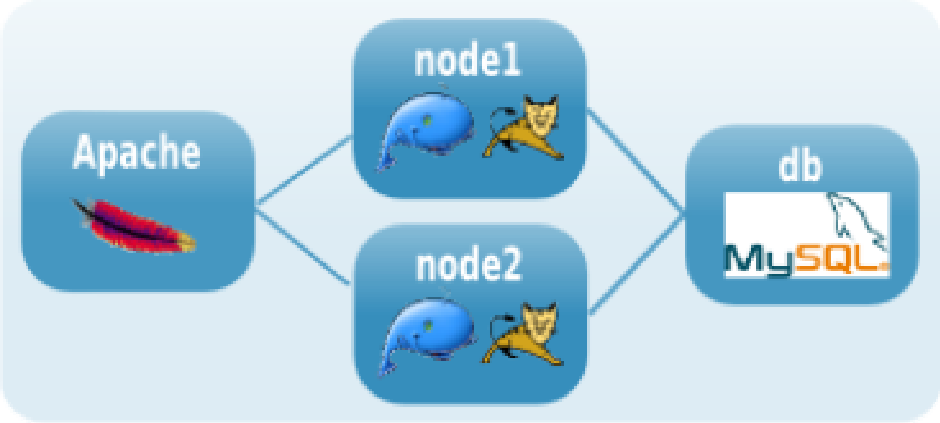



**figure 1. Cluster architecture**

### 4.1.2 Node hardware failure or server crash

This self-healing scenario aims at demonstrating how the autonomic system deals with fail-stop failure of the ME[1], eg. a node hardware failure or a Java EE middleware crash.

The autonomic manager allows the ME to recover from their failures by periodically monitoring the status of MEs via heart-beat sensors associated with the MEs at different levels (OS, JVM).

When a failure of a ME is detected, a repair mechanism is thrown. If needed, for instance after a hardware failure, a new node can be allocated and the middleware is installed. Afterwards the Java EE application server is configured as the failed one. The bindings with the front-end and back-end servers are recreated.

The following figure illustrates the scenario:

- Step1: nominal mode, the cluster works fine with 2 nodes for the application server tier: node1 and node2. An application is deployed across the 2 nodes and the Apache front end performs a load-balancing of the user requests.
- Step2: node1 crashes. The clustering mechanisms (fail-over and session replication) of the underlying middleware (JOnAS) ensure the continuity of service and availability of the user session.
- Step3: the autonomic manager detects the failure. A new node (node2) is gotten from the server pool.
- Step4: the autonomic manager deploys the middleware on the new node and configures its from its knowledge base.

[1] Managed Element defined in the Selfware's architecture document [3]

- Step5: the autonomic manager enables the new node. The links with the other components such as Apache at the front-end and MySQL at the back-end are activated.

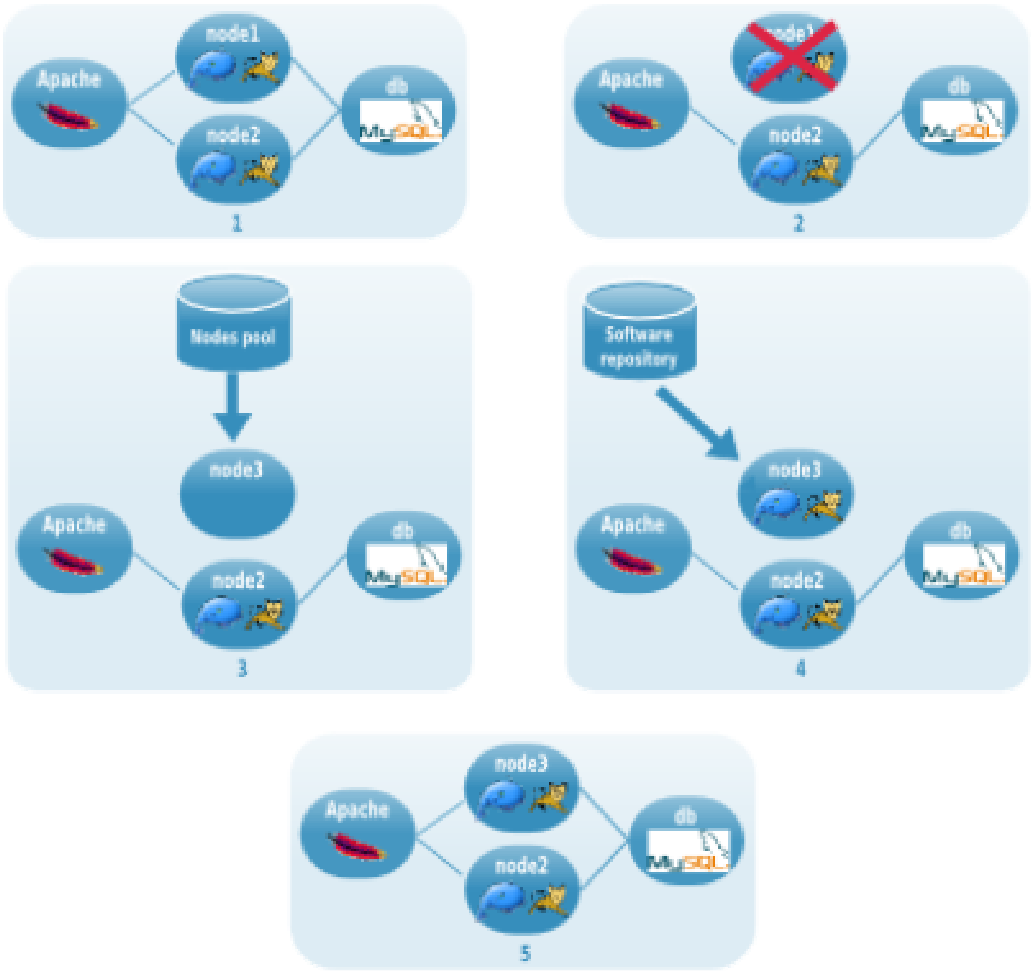



**figure 2. Server crash scenario**

### 4.1.3 Dynamic resource provisioning

This Self-Optimization scenario aims at maximizing application performance while minimizing the underlying resource usage (e.g. cluster nodes) through dynamic resource provisioning.

The autonomic manager targets MEs that represent a Cluster of replicated MEs[2], e.g. a cluster of JOnAS application server. The Cluster ME is periodically monitored via load sensors (e.g. CPU load, memory consumption, or an aggregation of sensors). 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.

The cluster resizing consists in first to allocate a new node and to install the Java EE middleware. Afterwards, one sub-ME of the cluster is introspected 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.

The following figure illustrates the scenario:

[2] The ‘cluster ME’ ter mis introduced in the Selfware’s architecture document [3]

- Step1: nominal mode, the cluster works fine with 2 nodes for the application server tier: node1 and node2. An application is deployed across the 2 nodes and the Apache front end performs a load-balancing of the user requests. The cluster load is monitored through probes on each ME.
- Step2: the cluster cpu load reaches the maximum threshold. An event is send to the autonomic manager.
- Step3: the autonomic manager decides to increase the cluster size. A new node (node3) is allocated from the server pool.
- Step4: the middleware is deployed and configured on the new node3.
- Step5: the cluster of 3 nodes is now operational and the global load in under the limit.
- Step6: the requests load decreases and thus the global load reaches the minimum threshold. An event is notified to the autonomic manager.
- Step 7: the autonomic managers decides to decrease the cluster size. The node3 is removed. Though the clustering mechanisms of the underlying middleware, there is no session loss and no interruption of service.
- Step 8: the cluster is come back to the initial size with 2 nodes..

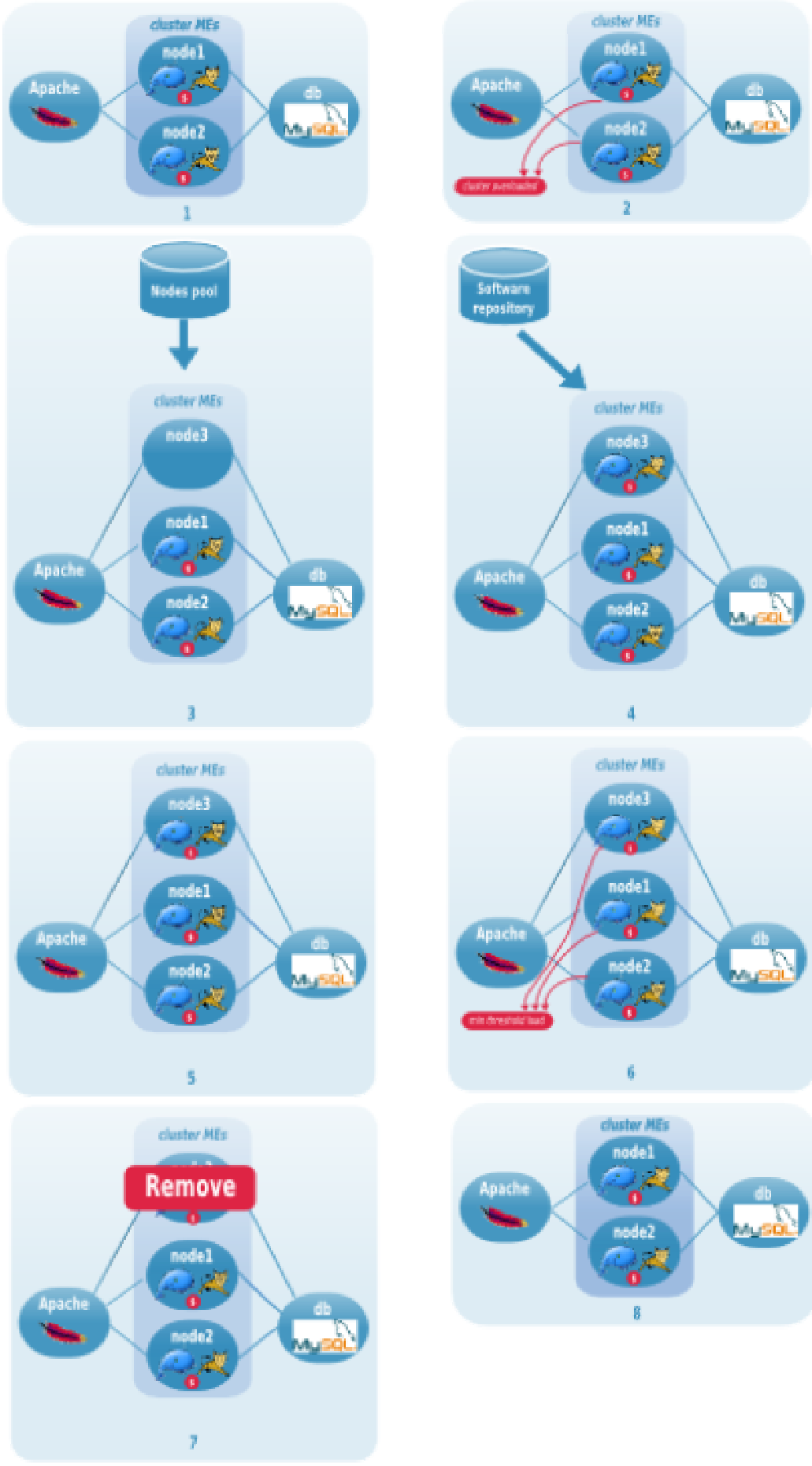

cluster MEs
node1
node2
Apache
db
1
2
Nodes pool
node3
3
Software repository
4
5
6
Remove
7
8

**figure 3. Self-sizing scenario**

### 4.1.4 Others scenarios

These scenarios are given for information and won't be implemented in the Selfware context.

#### Dynamic load-balancing

This Self-Optimization scenario aims at maximizing application performance by optimizing the underlying resource usage (ie. cluster nodes) through dynamic load-balancing tuning.

The autonomic manager targets MEs that belong to a Cluster of replicated MEs, ie. a cluster of JOnAS application server. Each ME is periodically monitored via load sensors (e.g. CPU load, memory consumption). When the load exceeds a given maximum threshold for a node while still being below for another node of the same Cluster ME, the load-balancing factors of each cluster node is adjusted for sending less requests to the overloaded node and more to the others.

#### Dynamic deployment

This Self-Optimization scenario aims at maximizing application performance by optimizing the deployment over a JOnAS cluster. The deployment optimization algorithm manipulates the application (Java EE module) and thus is finer grain that the 'Dynamic resource provisioning' scenario which relies on the more efficient JOnAS instances number for getting the better performance with the minimum of resources.

The autonomic manager targets MEs that belong to a Cluster of replicated MEs, ie. a cluster of JOnAS application server. Each ME is periodically monitored via load sensors (e.g. CPU load, memory consumption, Java EE application load). When the cpu load and the application load exceeds a given maximum threshold for a node while still being below for another node which doesn't host the same application the deployment topology is revisited. The number of nodes hosting the overloaded application is increased. Symmetrically, the number of nodes hosting the others application may be reduced.

#### Dynamic Selection of Component Implementations

This Self-Optimization scenario aims at maximizing application performance or improving QoS attributes by selecting the best Java EE component implementation at runtime.

Each component has multiple implementations, each one optimized for a certain execution context (IT resources, requests load, requests type, security, …).

## 4.2 Self-benchmarking

### 4.2.1 Introduction

### About benchmarking

The context of the scenarios proposed hereafter is that of load testing campaigns. Load testing campaigns consist in generating a flow of client requests on a system under test in order to assess its performance and sustainable throughput. As shown by figure 4, a load testing infrastructure is typically composed of:

- one or several load injectors sending requests to a system under test (SUT) and waiting for responses to measure the corresponding response times;
- probes measuring the usage of computing resources, at the SUT side, to help detecting performance problems, as well as at the load injection side, in order to check that it is performing as expected;
- a supervision user interface to deploy, control and monitor the distributed set of load injectors and probes;
- a storage space to gather all measures (e.g. as a set of log files or a database);
- tools for post-mortem analysis and report generation.

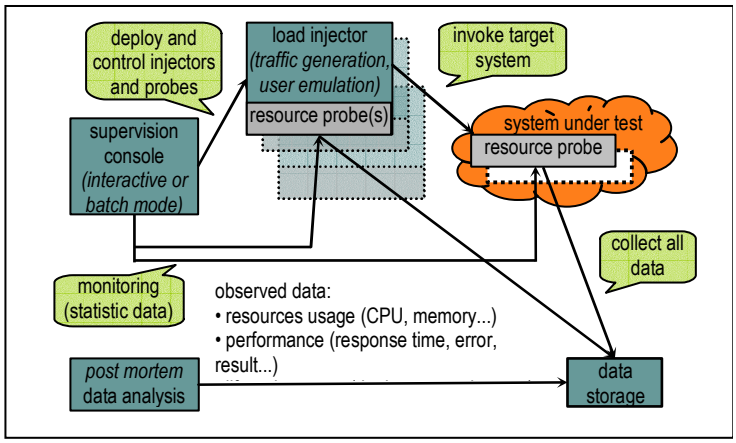



**figure 4. Big picture of a typical load testing infrastructure.**

The traffic generated by the load injectors are commonly modeled through the definition of virtual users, i.e. programs that emulate the behavior of real users, through successions of requests and think times (time spent by a user between 2 consecutive requests). For a given SUT, there are often a number of different typical usages, thus resulting in defining a number of different virtual users exhibiting different behaviors. In the case of web applications, some users may just consult available information, while others will strongly interact and trigger complex processes, resulting in different usage of computing resources and thus different impact on performance. For instance, some behaviors induce database write operations while others don't.

Performance benchmarking aims at comparing and ranking a variety of options such as configuration parameters or alternative implementations of hardware or software implementations, from a performance point of view. Benchmarking relies on load testing campaigns where the SUT must be tuned for optimal performance in order to obtain a meaningful ranking, since comparing results from an optimally configured SUT with results from a badly configured alternative would make no sense.

## Self-benchmarking

It typically takes a lot of manpower, skills and time to carry a load test campaign out. The test infrastructure is a complex combination of the load injection system, probe system, and SUT involving a tremendous number of parameters that are likely to strongly interact with each other (e.g. size of buffers, pools of database connections, size and policy of caches, network configuration, multi-threading policy...). Testers must be experts in every element of the global system (hardware, software, operating system, middleware, network equipments and protocols...) in order to handle troubleshooting and performance optimization. In an empirical and iterative process, tests are repeated again and again with different parameters arrangements and different configurations until sufficient confidence and satisfaction about results are met. Then, we see that testers behave like a feedback/control loop, observing the SUT and the load injection system on the one hand, and modifying the SUT and load injection configuration on the other hand as a reaction to observations.

Self-benchmarking and the scenarios detailed hereafter consist in considering that the tremendous complexity of the whole computing system used in a benchmarking campaign justifies an autonomic computing approach, that is: try to use computing power to autonomously deal with the computing system complexity. In other words, self-benchmarking shall carry out test campaigns by autonomously controlling the load injection system and the SUT configurations, with the objective of maximizing performance[3]. Self-optimization and self-configuration features are typically involved in this process.

### 4.2.2 Step one – autonomous search of saturation point

The first step in benchmarking generally consists in searching the approximate performance limit of the SUT in a given configuration. For example, in the context of application servers; the tester would typically try to find out the maximum number of users that the SUT may sustain with regard to given saturation criteria. Common criteria are expressed in terms of response time, request rejection, error occurrence, or computing resource shortage. A common way of looking for the saturation limit is to run a variable (generally growing) number of virtual users and look for the saturation point. Another way of varying the load injection is to change the proportion between the different virtual user families (i.e. of different behaviors). During this experimental search, the tester plays an empirical feedback role on the load injection system: according to the distance between the observation and the given saturation criteria, the tester more or less increases (or possibly decreases) the load.

Our first scenario is about defining, implementing and experimenting a load testing infrastructure featuring autonomous search of saturation point. As shown by figure 5, this infrastructure is composed of:

- the system under test, which will actually be a multi-tiers web application;
- the load injection system, made of one or several HTTP traffic generators depending on the required load level;
- probes measuring computing resources usage at the SUT, including for instance typical system probes (CPU, memory, network...) as well as possibly probes related to specific software elements involved in the multi-tiers environment (HTTP front-end, servlet server, EJB container, EJBs, database...);
- a load injection controller, replacing the {user interface, tester} pair by autonomously controlling the traffic level (number of virtual users) and/or contents (virtual user behavior), observing the overall system (load injection and SUT) through measures produced by probes (resource usage) and injectors (response times, errors, throughput), and controlling the load injection.

Then, it appears clearly that our load injection system involves a control loop, in conformance with Selfware's architectural approach to autonomic systems. Since the concept of saturation may be practically characterized in a number of ways, the architecture of our self-regulated injection system also exhibits a component dedicated to provide and isolate the saturation criteria and feedback (load injection) policy.

[3] Of course, the concept of performance may be practically mapped to a variety of criteria, such as request throughput, rejection or error rate, response times, number of users, etc.

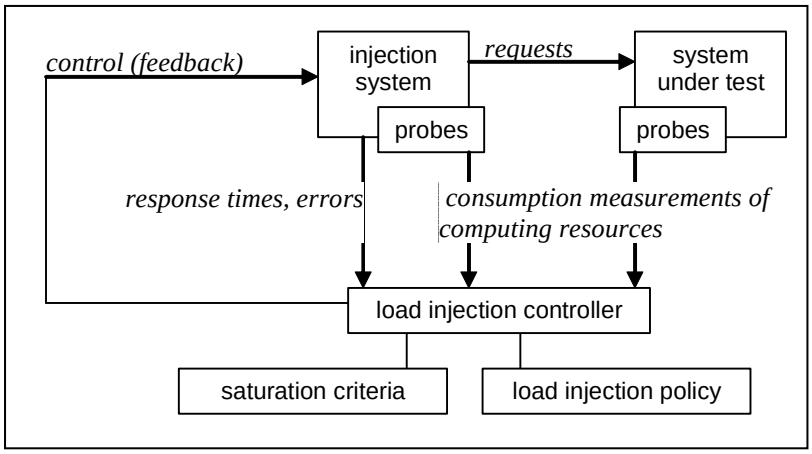



**figure 5. Self-regulated load injection for autonomic search of system performance saturation**

### 4.2.3 Step 2 – self-optimization of the system under test

This scenario is dedicated to the self-optimization of the tested system. As a matter of fact, the ultimate goal of benchmarking is to qualify the optimal performance of a system and to compare it to other similar but different (in configuration or implementation) system. Therefore, finding the optimal settings of the tested system, in other words optimize it, is key to the autonomic benchmarking principle.

This second step includes step one since benchmarking requires to reach the maximum performance (i.e. the saturation limit). For instance, the result of step one could be the conclusion that the SUT in a given configuration sustains a number of active virtual users representing a given mix of a number of behaviors. Then, the question is: is that result the best the SUT can deliver, or is it improvable by tuning the SUT? In other words, the tester will try to optimize the SUT and rerun the load test until s/he has the conviction that the maximum performance has been reached. This is, of course, a matter of estimation, whose accuracy depends on the skill and experience of the tester, since the combinatory of tuning parameters, and the complexity of interactions between them are so huge that a full exploration of the solution space is not humanly feasible.

This second scenario also consists in replacing the tester by a second control loop introducing a self-optimization of the SUT. Synchronization must be achieved between self-optimization and self-regulated load injection processes in order to alternate in a consistent way optimization phases and saturation search phases. The resulting architecture (see figure 6) adds a configuration controller, observing the maximum system performance (i.e. achieved at the saturation limit) reached for current configuration, and generating new possible system configurations. SUT-specific configuration rules must be provided in order to identify possible tunable parameters and their possible values. A general controller component orchestrates the configuration controller and the load injection controller.

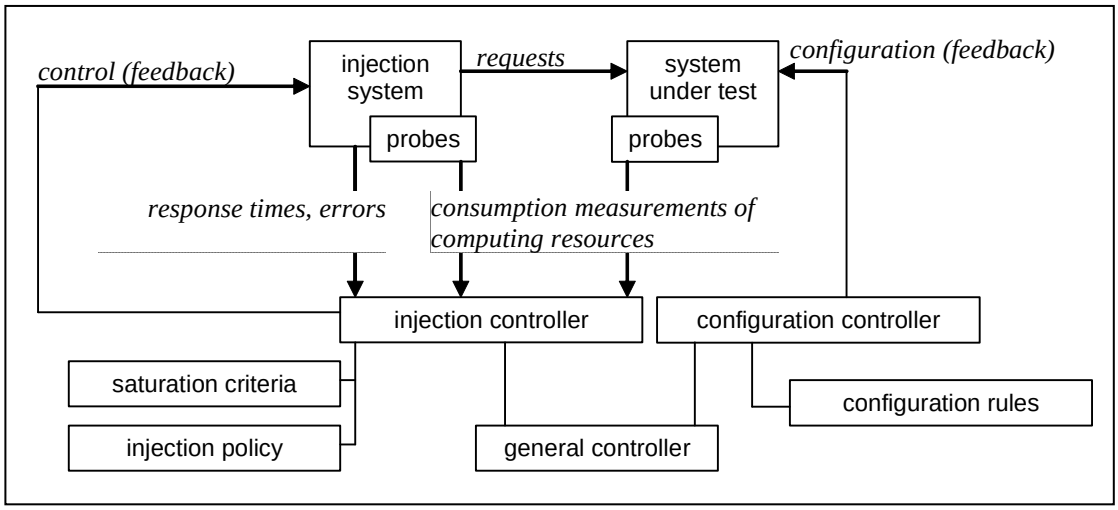
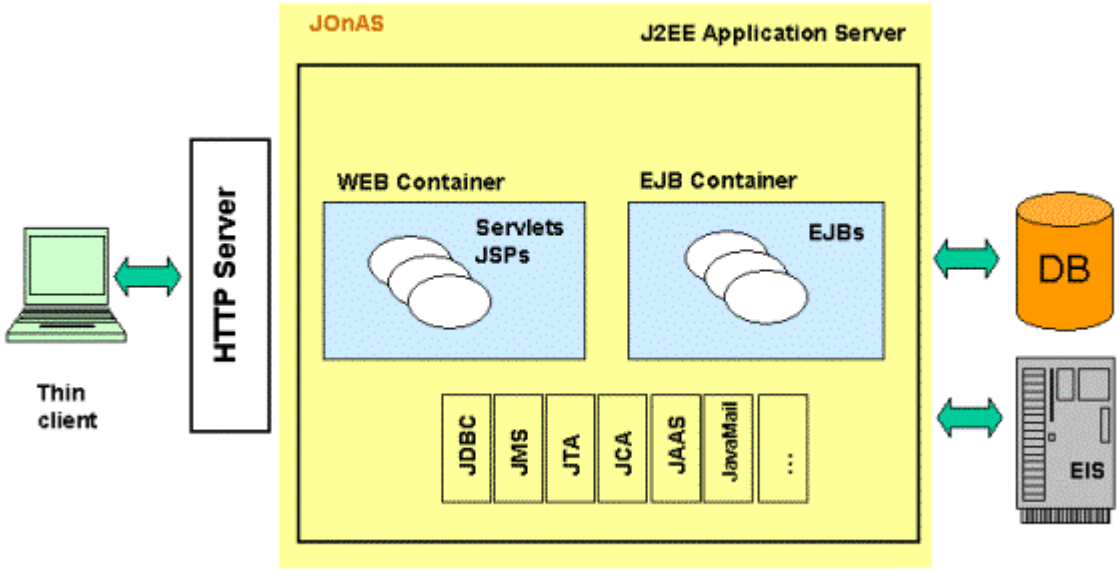



**figure 6. Autonomic benchmarking: self-optimization of a system under test and autonomic search of performance saturation**

# 5 Appendix

## 5.1 JOnAS, the Objectweb's application server

JOnAS is an open source application server, developed within the ObjectWeb consortium.




### 5.1.1 JOnAS Features

JOnAS is a pure Java, open source, application server. Its high modularity allows to it to be used as :

- a Java EE server, for deploying and running EAR applications (i.e. applications composed of both web and ejb components),
- an EJB container, for deploying and running EJB components (e.g. for applications without web interfaces or when using JSP/Servlet engines that are not integrated as a JOnAS container),
- a Web container, for deploying and running JSPs and Servlets (e.g. for applications without EJB components).

#### System Requirements

JOnAS is available for JDK 1.4 and JDK 5. It has been used on many operating systems (Linux, AIX, Windows, Solaris, HP-UX, etc.), and with different Databases (Oracle, PostgreSQL, MySQL, SQL server, Access, DB2, Versant, Informix, Interbase, etc.).

#### Java Standard Conformance

JOnAS supports the deployment of applications conforming to Java EE 1.4 specification. Its current integration of Tomcat or Jetty as a Web container ensures conformity to Servlet 2.4 and JSP 2.0 specifications. The JOnAS server relies on or implements the following Java APIs: EJB 2.1, JTATM 1.0.1, JDBC 3.0, Java EE CA 1.5, JMX 1.2, JNDI 1.2.1, JMS 1.1, JavaMail 1.3, Servlet 2.4, JSP 2.0, JAAS 1.0, JACC 1.0, Web Services 1.1, JAX-RPC 1.1, SAAJ 1.2, JAXR 1.0, Java EE Management 1.0, JAF 1.0, JAXP 1.2 specifications. JOnAS is architectured in terms of services.

## Key Features

JOnAS provides the following important advanced features:

- *Management*: JOnAS server management uses JMX and provides a JSP/Struts-based management console. It provides high level functionalities for monitoring and managing clusters of JOnAS servers.
- *Services*: JOnAS's service-based architecture provides for high modularity and configurability of the server. It allows the developer to apply a component-model approach at the middleware level, and makes the integration of new modules easy (e.g. for open source contributors). It also provides a way to start only the services needed by a particular application, thus saving valuable system resources. JOnAS services are manageable through JMX.
- *Scalability*: JOnAS integrates several optimization mechanisms for increasing server scalability. This includes a pool of stateless session beans, a pool of message-driven beans, a pool of threads, a cache of entity beans, activation/passivation of entity beans, a pool of connections (for JDBC, JMS, Java EE CA), storage access optimizations (shared flag, isModified).
- *Clustering*: JOnAS clustering solutions, both at the WEB and EJB levels, provide load balancing, high availability, and failover support.
- *Distribution*: JOnAS works with several distributed processing environments, due to the integration of the CAROL (Common Architecture for RMI ObjectWeb Layer) ObjectWeb project, which allows simultaneous support of several communication protocols:
  - RMI using the Sun proprietary protocol JRMP
  - RMI on IIOP
  - CMI, the "Cluster aware" distribution protocol of JOnAS
  - IRMI, an open source RMI protocol implementation
  - JOnAS benefits from transparent local RMI call optimization.
- *Support of "Web Services:"* Due to the integration of AXIS, JOnAS allows Java EE components to access "Web services" (i.e., to be "Web Services" clients), and allows Java EE components to be deployed as "Web Services" endpoints. Standard Web Services clients and endpoints deployment, as specified in Java EE 1.4, is supported.
- *Support of JDO*: By integrating the ObjectWeb implementation of JDO, SPEEDO, and its associated Java EE CA Resource Adapter, JOnAS provides the capability of using JDO within Java EE components.
- *Early support of EJB3*: Even before the EJB3 specification (part of Java EE 5, the new Java EE specification) is finalized, JOnAS already provides an EJB3 container. This one is available as a Java EE CA Resource Adapter, which may be deployed on JOnAS 4. It is developed as a standalone ObjectWeb project, named EasyBeans. This container makes EJB development far more easy.

Three critical Java EE aspects were implemented early in the JOnAS server:

- *Java EECA*: Enterprise Information Systems (EIS) can be easily accessed from JOnAS applications. By supporting the Java Connector Architecture, JOnAS allows deployment of any Java EE CA-compliant Resource Adapter (connector), which makes the corresponding EIS available from the Java EE application components. For example, Bull GCOS mainframes can be accessed from JOnAS using their associated HooX connectors. Moreover, with Java EE 1.4, resource adapters are now the "standard" way to plug JDBC drivers and JMS implementation, to Java EE platforms. A JDBC Resource Adapter is available with JOnAS, which provides JDBC PreparedStatement pooling and can be used in place of the JOnAS DBM service. A JORAM JMS Resource Adapter is also available. The JOnAS EJB3 early container is also available as a Resource Adapter.
- *JMS*: JMS implementations can be easily plugged into JOnAS. They run as a JOnAS service or through a resource adapter in the same JVM (Java Virtual Machine) or in a separate JVM. JOnAS provides administration facilities that hide the JMS proprietary administration APIs.

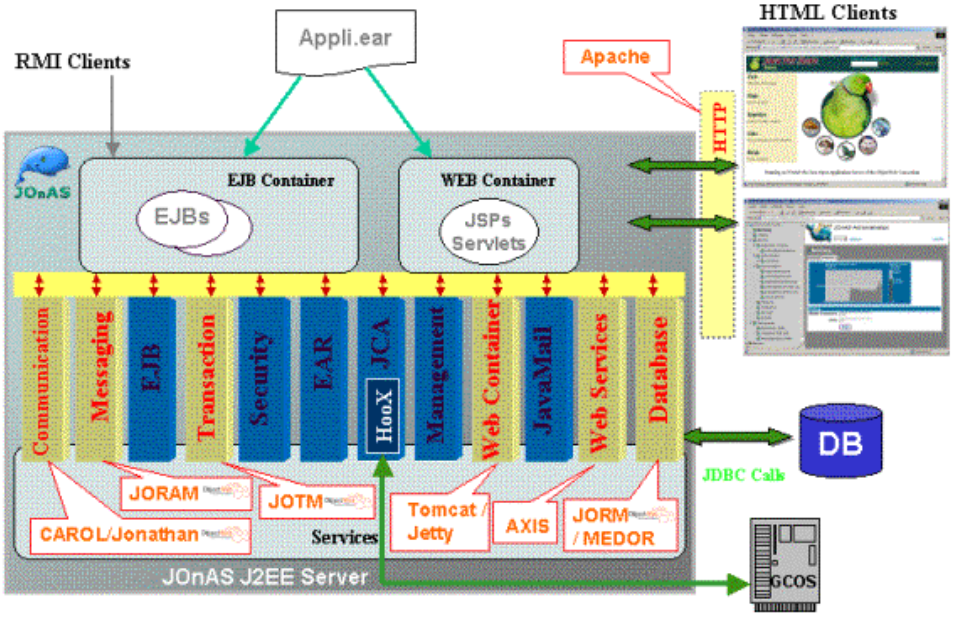
Currently, two JMS implementations can be used: the JORAM open source JMS implementation from Objectweb, and Websphere MQ. JORAM management is particularly strongly integrated within the JOnAS management console. Clustering configurations involving load balacing and high availability at the JMS level are also available thanks to JORAM HA features.

- *JTA*: The JOnAS platform supports distributed transactions that involve multiple components and transactional resources. The JTA transactions support is provided through JOTM, a Transaction Monitor that has been developed on an implementation of the CORBA Transaction Service (OTS). Originally this transaction monitor was JOnAS internal; it has been extracted to be available as a standalone project, JOTM.

### 5.1.2 JOnAS Architecture

JOnAS is designed with services in mind. A service typically provides system resources to containers. Most of the components of the JOnAS application server are pre-defined JOnAS services. However, it is possible and easy for an advanced JOnAS user to define a service and to integrate it into JOnAS. Because Java EE applications do not necessarily need all services, it is possible to define, at JOnAS server configuration time, the set of services that are to be launched at server start.

The JOnAS architecture is illustrated in the following figure, showing WEB and EJB containers relying on JOnAS services (this figure only misses the HA service). Two thin clients are also shown in this figure, one of which is the JOnAS administration console (called JonasAdmin).

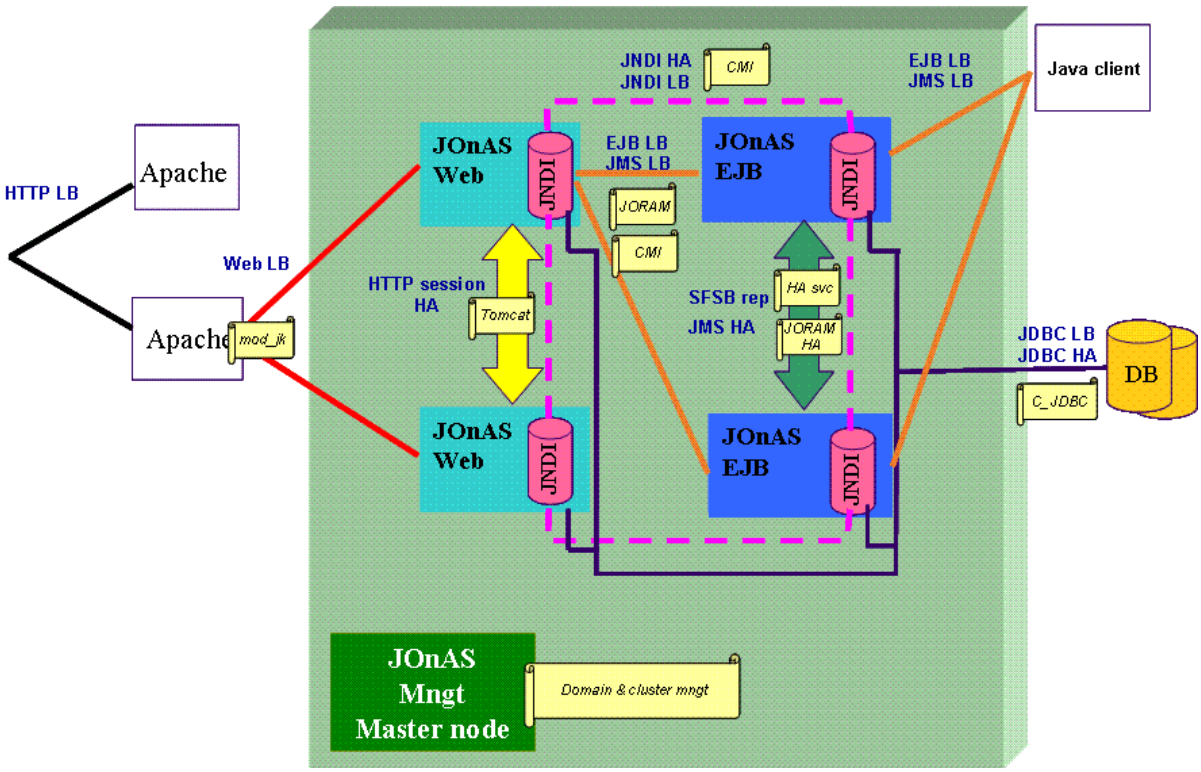


### 5.1.3 JOnAS clustering

The Java EE clusters aim at providing scalability and high availability to the applications. It relies on:

- A load balancing mechanism for distributing the requests over multiple servers and thus increase the processing capacities
- A replication mechanism of the essential data between the servers
- A transparent fail-over mechanism enabling to switch another server without service disruption, thus ensuring continuity of service.

To achieve these objectives, JOnAS provides a global solution for the clustering as illustrated in the following figure.




#### HTTP session load balancing

In a JOnAS cluster, load balancing at this level is achieved by dispatching HTTP requests from APACHE server(s) over a set of available JOnAS Web Containers (TOMCAT). As a result, more requests can be processed concurrently.

The MOD_JK plug-in provides communication between APACHE and TOMCAT servers.

#### HTTP session replication

In a general matter, HTTP session replication ensures service availability and continuity of service at the SERVLET or JSP level.

Session replication implies that the current service state - any data inside the HTTP session - is replicated across multiple application instances.

It is the responsibility of the application to choose which data to store inside the session to ensure that a failover happens transparently without a disruption of the entire solution.

In a JONAS cluster, the session replication, called all-to-all replication, happens inside the Web Containers (TOMCAT) that use a proprietary TCP based protocol.

### JNDI clustering

The CMI protocol provided with JOnAS, embeds its own replicated registry. The fat clients can use a servers list in the lookup requests ensuring both the load-balancing and the high availability of the JNDI accesses.

### EJB load balancing

In a general matter, load balancing at this level is achieved by dispatching EJB invocations from the clients over a set of available EJB Containers. As a result, more requests can be processed concurrently.

For a Stateless Session Bean (SSB), the load balancing takes place for either the home or the business methods (remote interface). For a State Full Session Bean (SFSB), the load balancing takes place only for the home interface.

In a JOnAS cluster environment, the cluster has knowledge, through the JGroups protocol, of the distribution of the EJBs.

The CMI protocol, implemented on top of RMI, supports EJB invocations for fat Java clients or Web applications running within the Web containers.

An answer to a lookup (or create for the SSB) is a special stub containing stubs to each instance known in the cluster. The stub can issue each method call on the home (remote for SSB) of the EJB to a new instance and uses a weighted round robin algorithm for distributing the load.

### Session Full EJB replication

This solution enables failover at EJB level.

This means state replication for State Full Session Beans (SFSB).

Only the state of the EJB is replicated, not the bean itself. The "state" contains enough information for the nodes to be able to re-create the bean.

An Entity Bean (EB) cannot be replicated. However, the state of an SFSB can contain a reference to an Entity Bean (EB).

### JMS clustering

Through JORAM, the JMS provider integrated with JOnAS, and its distributed architecture, the JMS objects requests can be balanced among several JORAM servers. Furthermore, fail-over is provided with the JORAM HA relying on one master/N slaves architecture

### JDBC clustering

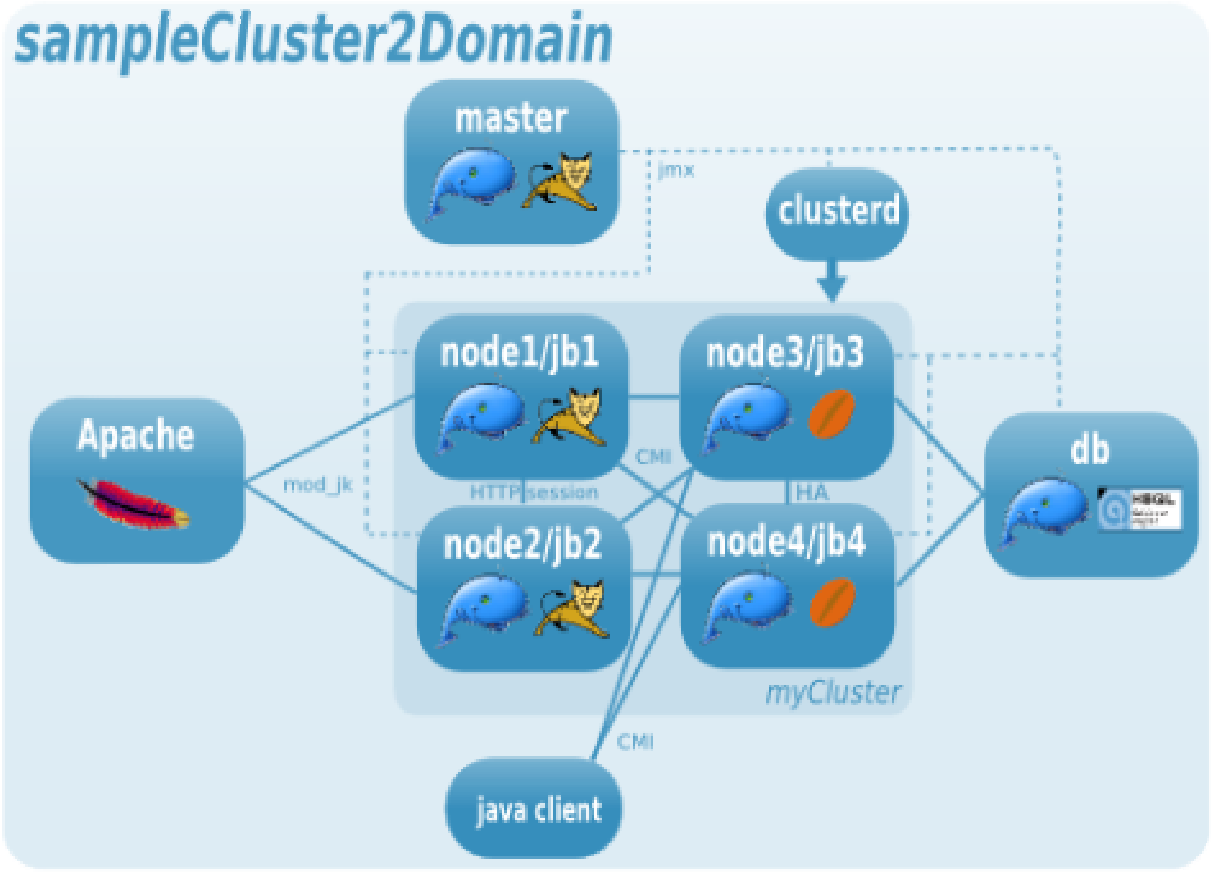
Through the Sequoia project (formet ObjectWeb C_JDBC project) the writing JDBC requests are multicasted to several database nodes ensuring high availability and the reading JDBC requests are load-balanced. Furthermore, an extra cache layer (C-JDBC controller) increases the reading performances.

### Clustering management

The jonasAdmin console enables to perform cluster operations such as deployment or monitoring from a management dedicated node called the master node.

### Clustering architecture example

An example of a typical JOnAS cluster architecture is given in the following figure.




- *master node* : management node enabling to administrate the cluster from a centralized console
- *db node* : database node, here it's is a HSQL instance embedded into a JOnAS
- *cd* : cluster daemon, bootstrap of a JOnAS node for the remote control
- *apache/mod_jk* : HTTP server with the plugin enabling the load balancing of the HTTP flow
- *node1-2* : JOnAS nodes of the web level
- *node3-4* : JOnAS nodes of the ejb level
- *myCluster* : logical cluster gathering the 4 JOnAS nodes, described in domain.xml
- *sampleCluster2Domain* : domain name of the configuration
- *java client* : fat client
- *CMI* : RMI protocol for the cluster mode ensuring the JNDI replication and the EJB clustering
- *HA* : SFSB replication
- *HTTP session* : HTTP session replication

# 6 References

1 Ada Diaconescu, Automatic Performance Optimisation of Component-Based Enterprise Systems via Redundancy, 2006

2 Sara Bouchenak, Fabienne Boyer, Noel De Palma, Daniel Hagimont, Sylvain Sicard, and Christophe Taton, JADE: A Framework for Autonomic Management of Legacy Systems, 2006

3 Selfware Architecture, 2007