TAPAS

IST-2001-34069

Trusted and QoS-Aware Provision of Application Services

Using the JBoss Clustering Service in the TAPAS Platform

Report Version: 0.1

Report Delivery Date: (this is not a deliverable)

Classification: Internal report. Draft for comments only. Please do not circulate.

Contract Start Date: 1 April 2001 **Duration:** 36m

Project Co-ordinator: Newcastle University

Partners: Adesso, Dortmund – Germany; University College London – UK; University of Bologna – Italy; University of Cambridge – UK

Project funded by the European Community under the "Information Society Technology" Programme (1998-2002)

Using the JBoss Clustering Service in the TAPAS Platform

Giorgia Lodi, Fabio Panzieri
University of Bologna
Department of Computer Science
Mura A. Zamboni 7
I – 40127 Bologna
{lodiglpanzieri}@cs.unibo.it

Abstract

The JBoss application server provides its users with a clustering service that implements load balancing and failover mechanisms, within a JBoss cluster. In this report we discuss the use of this service in the implementation of the TAPAS Platform.

Introduction

The TAPAS platform incorporates a so-called Configuration Service [1], responsible for constructing, and maintaining at run time, the hosting environment within which distributed applications can be deployed and run.

For the purposes of this Report, we shall assume that an Application Service Provider (ASP) provide its customers with a complete application hosting environment which can fully host the applications those customers wish to run (i.e., we assume a possibly simplistic scenario in which no additional providers, such as an Internet Service Provider or a Storage Service Provider, are involved). This hosting environment may consist of a cluster of interconnected machines, governed by possibly heterogeneous operating systems; each machine in that cluster runs an instance of the JBoss application server.

Within this scenario, the TAPAS Configuration Service exercises control over both the internal configuration of each application server instance, in the application hosting environment, and the set of clustered server instances that form this environment.

Hence, this Service can be thought of as operating at two distinct levels of abstraction, that we term *micro-resource* and *macro-resource* levels, respectively. The former level consists of resources, such as server queues and thread pools, internal to each individual application server; the latter level consists of such resources as the server instances that form the application hosting environment.

Thus, for example, the Configuration Service at the micro-resource level may have to adjust dynamically an application server queue length, in order to allow that server to deal with particularly demanding load conditions, and to maintain its responsiveness. In contrast, in order to meet possible load balancing and responsiveness requirements at the macro-resource level, the Configuration Service may have to extend the cluster configuration by enabling a new application server instance, or to replace a crashed application server instance with an operational one, at application run time.

In this Report we discuss the use of the JBoss clustering service in the implementation of those Configuration Service functionalities operating at the macro-resource level.

This Report is structured as follows. The next Section summarizes the principal features of the JBoss clustering service. Section 3 describes how those features are related to the implementation of the TAPAS Configuration Service, and discusses some implementation issues. Finally, Section 4 concludes this Report.

2 JBoss Clustering Service

A JBoss cluster (or partition) is defined as a set of *nodes*. A node, in JBoss, is a JBoss application server instance. There can be different partitions on the same network; each partition is identified by an individual name. A node may belong to one or more partitions (i.e., partitions may overlap); moreover, partitions may be further divided into *sub-partitions*. Partitions are generally used for scopes of load distribution purposes, whereas sub-partitions may be used for fault-tolerance purposes (it is worth mentioning that, to the best of our knowledge, there are no implementations of the sub-partition abstraction, as of today).

A JBoss cluster can be used for either *homogeneous* or *heterogeneous* application deployment. Homogeneous deployment entails that each node in the cluster runs identical services, and Enterprise Java Beans (EJBs); in contrast, heterogeneous deployment entails that each node in the cluster may run a different set of services and EJBs. It is worth observing that, in practice, this latter form of clustering is not recommended [2]; hence, for purposes of our current discussion, in the following we shall assume homogeneous deployment, only.

The JBoss Clustering service [3] is based on the framework depicted in Figure 1 below. This framework consists of a number of hierarchically structured services, and incorporates a reliable group communication mechanism, at its lowest level. The current implementation of this mechanism uses JGroups [4], a toolkit for reliable multicast communications. This toolkit consists of a flexible protocol stack that can be adapted to meet specific application requirements. The reliability properties of the JGroups protocols include lossless message transmission, message ordering, and atomicity.

Specifically, JGroups provides its users with reliable unicast and multicast communication protocols, and allows them to integrate additional protocols (or to modify already available protocols) in order to tune the communication performance and reliability to their application requirements. JGroups guarantees both message ordering (e.g., FIFO, causal, total ordering), and lossless message transmission; moreover, a message transmitted in a cluster is either delivered to each and every node in that cluster, or none of those nodes receives that message (i.e., it supports atomic message delivery). In addition, JGroups enables the management of the cluster membership, as it allows one to detect the starting up, leaving and crashing of clustered nodes. Finally, as state transfer among nodes is required when nodes are started up in a cluster, this state transfer is carried out maintaining the cluster-wide message ordering.

The HighAvailable Partition (HAPartition) service is implemented on top of the JGroups reliable communications; this service provides one with access to basic communication primitives which enable unicast and multicast communications with

the clustered services. In addition, the HAPartition service provides access to such data as the cluster name, the node name, and information about the cluster membership, in general. Two categories of primitives can be executed within a HAPartition, namely state transfer primitives, and RPCs.

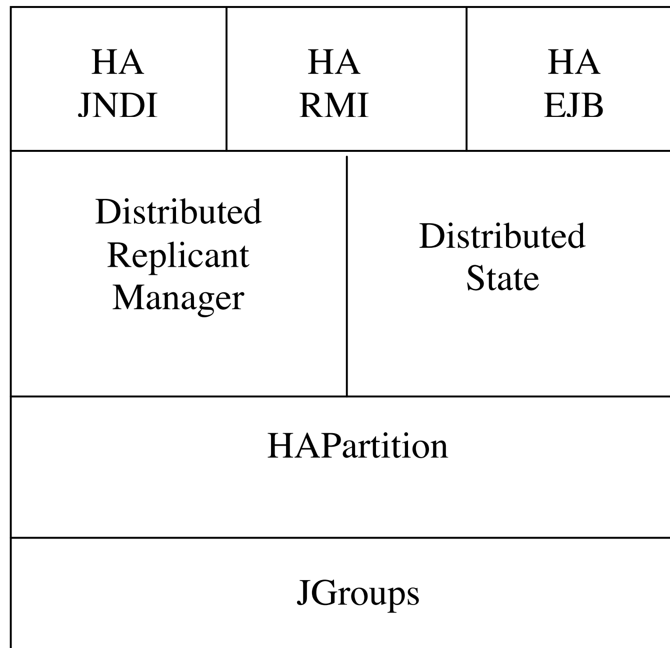

Fig. 1: JBoss Clustering Framework

The HAPartition service supports the Distributed Replicant Manager (DRM), and the Distributed State (DS) services. The DRM service is responsible for managing data which may differ within a cluster. Examples of this data include the list of stubs for a given RMI server. Each node has a stub to share with other nodes. The DRM enables the sharing of these stubs in the cluster, and allows one to know which node each stub belongs to.

The DS service, instead, manages data, such as the replicated state of a Stateful Session Bean, which is uniform across the cluster, and supports the sharing of a set of dictionaries in the cluster. For example, it can be used to store information (e.g., settings and parameters) useful to all containers in the cluster.

The highest JBoss Clustering Framework level incorporates the HA-JNDI, HA-RMI, and HA-EJB services. The HA-JNDI service is a global, shared, cluster-wide JNDI Context used by clients for object look up and binding. It provides clients with a fully replicated naming service, and local name resolution. When a client executes an object lookup by means of the HA-JNDI service, this service firstly looks for the object reference in the global context it implements; if the reference is not found, it requires the local JNDI to return that object reference. The HA-RMI service is responsible for the implementation of the smart proxies of the JBoss clustering (see next Section).

Finally, the HA-EJB service provides mechanisms for clustering different types of EJBs; namely, the Stateless Session Beans, the Stateful Session Beans, and the Entity Beans (no clustered implementation of the Message Driven Beans is currently available in JBoss 3.x). The cluster version of the Stateless Session Beans appear to be easy to manage, as no state is associated to those beans; the state management of the Stateful Session Beans is implemented. In contrast, the state management of the Entity Beans in a cluster is a rather complex issue which is currently addressed at the level of the database the Entity Beans interface, only (see Section 3).

2.1 Load Balancing and Failover in JBoss

The JBoss Clustering service implements load balancing of RMIs, and failover of crashed nodes (i.e., when a clustered JBoss node crashes, all the affected client calls are automatically redirected to another node in the cluster).

The implementation of the load balancing and the failover mechanisms in the JBoss Clustering Service can be based on one of the following three alternative models [6], depicted in Figure 2 below, and summarized in the following.

1. Server Based: the load balancing and the failover mechanisms are implemented on each clustered JBoss node;
2. Intermediary Server: these mechanisms are implemented by a proxy server;
3. Client based: these mechanisms are incorporated in the client application itself (in the RMI stub).

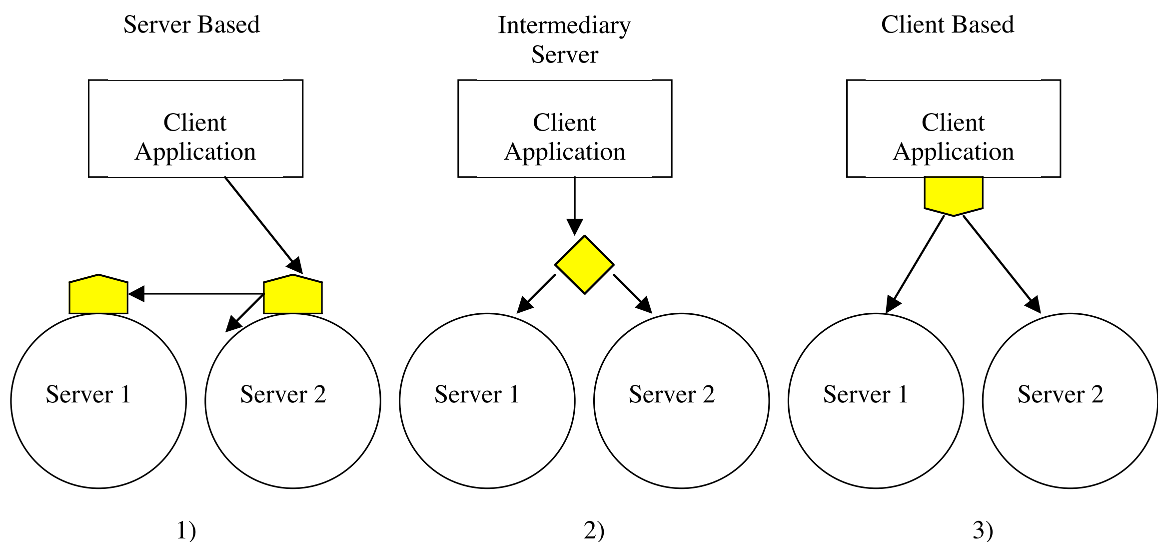

Fig. 2: Clustering Implementation Models

JBoss adopts the above model 3), which includes the load balancing and failover mechanisms inside the client stub. Specifically, a client gets references to a remote EJB component using the RMI mechanism; consequently, a stub to that component is downloaded to the client. The clustering logic, including the load balancing and failover mechanisms, is contained in that stub. In particular, the stub embodies both

the list of target nodes that the client can access, and the load balancing policy it can use.

Moreover, if the cluster topology changes, the next time the client invokes a remote component, the JBoss server hosting that component piggybacks a new list of target nodes as part of the reply to that invocation. The list of target nodes is maintained by the JBoss Server automatically, using JGroups. Thus, in general, following a client RMI, the client stub receives a reply from the invoked server, unpacks the list of target nodes from that reply, updates the current list of target nodes with the received one, and terminates the client RMI.

This approach has the advantage of being completely transparent to the client. The client just invokes a method on a remote EJB component, and the stub implements all the above mechanisms. From outside, the stub looks like the remote object itself; it implements the same interface (i.e., business interface), and forwards the invocations it receives to its server-side counterpart. When the stub's interface is invoked, the invocation is translated from a typed call to a de-typed call.

For instance, if the client calls the following method on a remote object:

```
myRemoteComponent.businessMethod(params);
```

this code will be transformed into the following system-level invocation:

```
proxyClientContainer.invoke(invocation);
```

where `invocation` is an instance of the `Invocation` class which contains (i) the arguments passed to the method, (ii) the method being called, and (iii) arbitrary payloads that can be added to the invocation [6]. The de-typed invocation is passed through a set of client-side interceptors, as depicted in Figure 3. The load balancing and failover mechanisms are located in the last interceptor of the chain (i.e., interceptor C in Figure 3 below).

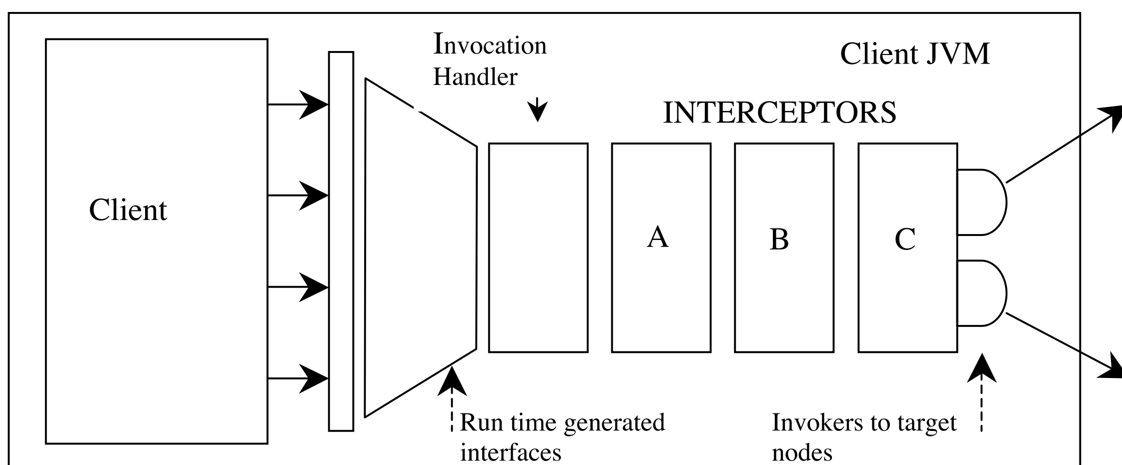

Fig. 3: Client-side interceptors

This interceptor uses the load balancing policy selected at deployment time in order to elect a target node where to forward the invocation.

Currently, JBoss 3.2 implements the following four load balancing policies, which can be specified into the EJB deployment descriptors:

Random Robin: each call is dispatched to a randomly selected node.

Round Robin: each call is dispatched to a new node. The first target node is randomly selected from the target node list;

First Available: each stub elects one of the available target nodes is as its own target node for every call (this node is chosen randomly). When the list of the target nodes changes, a new target node is elected only if the earlier elected one is no longer available.

First Available Identical All Proxies: this policy is the same as the *First Available* policy above. However, the elected target node is shared by a *proxy family*; i.e., a set of stubs that direct invocations to the same target node.

3 TAPAS Configuration Service

The TAPAS Configuration Service is responsible for setting up the platform required for hosting a distributed application (e.g., an electronic auction application); this platform is to be set up so as to meet effectively the QoS requirements of that application. Those requirements are specified in a Service Level Agreement (SLA), agreed upon by the application owner and the ASP; hence, the Configuration Service receives in input an SLA in order to start-up its platform configuration activity.

At start-up time, the Configuration Service interrogates a set of sensors, located in each resource of the clustered machines, in order to discover the availability of those resources. Its principal responsibility is to configure the application within the hosting environment running in those machines, at application deployment time, and to reconfigure the application at run time, if necessary. Specifically, at deployment time, the Configuration Service selects the most suitable cluster configuration that can meet the input SLA. At run-time, instead, it may have to reconfigure the cluster as a response to a reconfiguration request from the Controller Service [1] (e.g., in case this latter Service has detected variations in the hosting environment, which may lead to violations of the input SLA). This reconfiguration may either entail distributing the computational load across the clustered resources, dynamically, or extending the cluster with additional resources, if available (or both).

It is worth observing that a variety of independent applications may be hosted concurrently within the same physical cluster of resources, at least in principle. Each of these applications may have a different SLA with its hosting environment. In this context, a cluster-wide load balancing policy may well trade possible economic penalties, which may be caused by the violation of a specific application SLA, for resources required by another application (e.g., as the penalties incurred in violating the SLA of the former application are less dramatic than those which would be incurred in by violating the SLA of the latter application; as the revenues from the latter application are far superior than the former application penalties). Hence, the

operation of the above mentioned load balancing policy is to be based on a cluster-wide view of the state of the clustered resources.

The load balancing policies currently included in the JBoss clustering service are defined at deployment time, inside the EJB deployment descriptors. In particular, a load balancing policy can be specified for each bean, for the home and remote proxies, as shown in Figure 4, below.

```
<jboss>
<enterprise-beans>
  <session>
     <ejb-name>MySessionBean</ejb-name>
     <clustered>True</clustered>
     <cluster-config>
       <partition-name>DefaultPartition</partition-name>
          <home-load-balance-policy>
          org.jboss.ha.framework.interface.RoundRobin
          </home-load-balance-policy>
          <bean-load-balance-policy>
          org.jboss.ha.framework.interface.FirstAvailable
          </bean-load-balance-policy>
     </cluster-config>
  </session>
</enterprise-beans>
</jboss>
```

Figure 4: Deploying load balancing policies in JBoss

Note that load balancing policies currently available in JBoss implement non-adaptive load balancing within the cluster; hence, at run time, they may select a target node in the cluster which may well be overloaded or close to being overloaded. This limitation cannot be overcome by those policies as they operate with no knowledge of the effective load of the clustered machines, at run time.

However, additional load balancing policies can be incorporated in a JBoss application server by plugging them into the last interceptor of the chain, as illustrated in Figure 3. In particular, the load balancing in JBoss can be augmented with adaptive strategies that select the target machines at run time, based on the actual computational load of those machines.

3.1 Implementation Issues

The Configuration Service can be implemented as an Mbean, and integrated into the JBoss application server through the JMX software bus. It will be a pure server-side component; its implementation model is as depicted in the earlier Figure 2.1). We believe that this solution can be as transparent to the client code as that currently provided by the JBoss application server (depicted in Figure 2.3), and described earlier).

The operational principles of the Configuration Service we are implementing can be summarized as follows. Assume that a JBoss cluster be set up, and that homogeneous application deployment is to be carried out in order to meet the high availability requirements of an application to be run in that cluster.

Each JBoss node in the cluster incorporates a Configuration Server (CS) (i.e. an implementation of our Configuration Service), identified by a cluster-wide unique identifier (ID). Note that the management of the unique server ID's can be easily implemented using the JGroups view management protocol; we are not going to discuss this issue further in this Report.

At application deployment time, the CS with the lowest ID becomes the Configuration Leader. This Leader examines the application SLA, and contacts its peer CSs in the cluster in order to construct a suitable partition of nodes that can host the application.

Note that the nodes in that partition will host identical instances of the application, as homogeneous deployment is being carried out. Owing to the same motivation, with the current JBoss implementation of the clustering service, each node in that partition should be capable of honouring the application SLA.

As the partition is started up, the application can be deployed and run. Clients can issue RMIs to any node in the partition, transparently.

If a failure occurs, (e.g., the crash of a JBoss node in the partition), the standard failover mechanism in JBoss redirects the client RMIs, addressed to the crashed node, to another active node in the partition. This node will be selected according to one of the four load balancing policies introduced earlier, and specified at deployment time. As these policies select a target node with no knowledge of the run time computational load of that node, it is possible that the RMI redirection following a node failure in a partition lead to overloading another node in that partition. Note that, in principle, this process may continue until all nodes in that partition are brought to an overloaded state, as a sort of domino effect.

In order to overcome this problem, in our implementation the Configuration Service aims to maintaining a fair distribution of the computational load among the nodes belonging to the same partition. To this end, in case a node failure occur within a partition, our Configuration Service firstly attempts to reconfigure that partition by integrating in it a spare node that replace the faulty one; that spare node can be obtained possibly from another partition (or from a pool of resources reserved for this purpose, for example). Secondly, if no spare node is available, and the above reconfiguration cannot be carried out, the Configuration Service redistributes the RMIs addressed to the faulty node among the remaining nodes in the partition (this load distribution can be carried out using an adaptive load balancing policy that takes its decisions based on the current computational load of those nodes).

State consistency among the nodes belonging to the same partition can be maintained using the JGroup reliable communication framework, as JBoss currently does.

To conclude this session, we wish to point out that the Configuration Service introduced above is transparent to both the application client code, and the client stubs. In addition, it offers a higher degree of fairness, among the nodes belonging to the same partition, than the current JBoss failover mechanism.

3.2 Open Issues

Issues of heterogeneous application deployment, and application component replication can play an important role in the implementation of our Configuration Service. In this Subsection, we introduce these two issues, and examine three alternative implementations of our Service.

Heterogeneous deployment of application components consists of the ability of distributing the components of an application across a cluster of machines, in a controlled manner.

As already mentioned, the JBoss documentation available to us does not recommend the use of this form of application deployment, as there are a number of as yet unsolved problems related to it, including lack of i) distributed locking mechanisms for use from entity beans, and ii) cluster-wide configuration management.

However, we believe that heterogeneous deployment, in contrast with the homogeneous one discussed earlier, can be particularly attractive when applied to component-based technologies. Typically, these technologies adopt a distributed multi-tier paradigm, in which the application consists of separate components; namely, web components, and EJB components. In general, the Web components of an application are directly exposed to the clients, so as to mask the business tier in which the EJB components are located.

As we are considering a scenario in which client-server communications are enabled via wide area networks, as clients can be located geographically far away from the servers, it may well be convenient to distribute the application so that its Web components are as close as possible to the clients. Moreover, in order to reduce the application response times, it can be desirable to distribute the application EJB components so that those directly connected to the database (e.g. the entity beans) are located as close possible to the clustered database servers. This can be done both at deployment time, when the Configuration service acquires the application QoS requirements (i.e. the SLA), and at run time, when the application SLA is close to being violated (as reported by the Controller Service [1]).

Component replication is a further issue that deserves attention. It is worth distinguishing between replicating application components (i.e., EJB classes, Web component classes, and so on), and replicating instances (i.e., run-time data) of those components [7]. As to the first form of replication, JBoss provides a mechanism that automatically replicates components classes deployed on one node to the other nodes in the cluster. This mechanism is termed *Farming*. Instead, in order to support the latter form of replication, the JBoss clustering service provides [8]:

- Replicated state for Stateful Session Beans
- Replicated HTTP Sessions
- Replicated Entity Beans
- Global cluster-wide, replicated JNDI tree (HA-JNDI)

The Stateless Session Beans do not need to be replicated as no state is associated to them; they only need to be deployed within the cluster.

Owing to the above observations, three different implementation approaches of the Configuration Service suggest themselves; namely, a first approach implementing HEterogeneous Application Deployment (HEAD), a second one implementing a HOmogeneous Application Deployment (HOAD), and finally a third one implementing both forms of Deployment (HHAD). These three approaches are introduced below in isolation.

HEAD – In order to implement heterogeneous deployment, the Configuration Service has to know in detail the Deployment Descriptors (DDs) of all the application components, at deployment time, so as to optimize the physical distribution of those components (i.e., if components communicate by means of local interfaces, they must be located in the same JVM, and are to be deployed so as to ensure that the SLA is met). At run time, it can be possible to migrate components from overloaded machines to other, more lightly loaded, machines.

The principal advantage of this approach is that the heterogeneous deployment allows the Configuration Service to distribute the computational load so as to optimise the use of the available resources in the cluster. However, this approach requires that the Configuration Service know all the application component DDs , in order to distribute those components. Moreover, migration of components from one machine to another is a complex task that requires that issues of state propagation, distributed locking, and management of a cluster-global JNDI tree be carefully dealt with (the latter issue is addressed by the latest JBoss release).

Finally, we have already pointed out that, currently, JBoss implements non-adaptive load balancing policies. In contrast, we require the use of adaptive load balancing. Thus, we can either (i) integrate a new and adaptive load balancing policy into the JBoss clustering, or (ii) require that the Configuration Service implement its own adaptive load balancing. In the either cases, the load balancing policy distributes the computational load among the clustered machines, based on the actual load of those machines. However, the former implementation requires that the load balancing policy be specified in the EJB DDs, in order to be applied. In contrast, the latter implementation removes this requirement; however, it requires that new invokers be implemented on both the client and server sides.

HOAD – If support for homogeneous deployment is required (i.e. copies of the entire application are located in every machine, or in a sub-set of machines, in the cluster), the Configuration Service has to establish at deployment time the most suitable partition in which the application can be run. In this case, the entire application can be deployed in only one node of the cluster. Then, the JBoss Farming service provides its distributed deployment. At run time, if the hosting environment conditions change and the SLA is about to being violated, the Configuration Service has to either choose a different partition, or create a new one, and reconfigure appropriately the application..

This solution seems to be simpler than the first one: we just have to use the JBoss clustering framework, which provides us with the features we need, in this case. However, the problem related to the non adaptivity of the load balancing policies currently made available by JBoss still remains unsolved.

HHAD – Finally, the two approaches above can be combined as follows. At deployment time, the Configuration service can execute homogeneous deployment by replicating component classes using the JBoss Farming service; thus, it will create a partition in which the application can be run, and load balancing applied. At run time, if the SLA is close to being violated, the re-configuration activity of the Configuration Service can decide whether to (i) add more replicas of the entire application, (ii) add a certain number of machines to the initial cluster partition, i.e. to augment the number of available resources, (iii) migrate application components from overloaded machines either to other ones in the current partition, or different machines that do not belong to this partition.

The third option may lead to having different beans of the same application deployed and running onto different machines (i.e., it may lead to heterogeneous application deployment). However, implementing this option may have a notable impact on the overall performance of the clustered application server, as migrating application components within the cluster can be very costly. Hence, this option can be used only when some particular hosting conditions occur (e.g. critical thresholds are reached within the hosting environment, and detected during the SLA monitoring phase).

Finally, even in this third option, we can use the JBoss clustering features; however, some changes to the current JBoss implementation are required in order (i) to apply an adaptive load balancing policy, (ii) to provide a distributed transactional manager, and (iii) to both manage migration operations, and improve the performance of those operations.

A summary of this discussion is contained in table 1, below.

4 Concluding Remarks

In this Report we have examined the JBoss clustering features, in the view of their use in the implementation of the TAPAS platform Configuration Service. Specifically, we have examined the current implementation of the JBoss clustering service, and suggested three different approaches to the implementation of our Configuration Service.

Our discussion of the JBoss clustering service has shown that the principal shortcoming of this service is that it does not provide its users with an adaptive load balancing policy (although it does not prevent those users from implementing their own policies).

As we believe that this form of load balancing is required within the TAPAS platform, we have proposed the integration of one such policy in the application server; in particular, in this Report we have proposed to impelment an adaptive load balancing strategy at the server level (rather than at the client level, as in the current JBoss implementation).

In addition, in this Report we have pointed out the following limitations we have found in the current JBoss clustering implementation:

- A cluster (i.e., all its nodes) is (are) used completely when a clustered application is homogeneously deployed; i.e., there is no a dynamic Farming Service. For instance, if the cluster is composed by five nodes (i.e. five JBoss server instances), the application components cannot be deployed in a sub-set of machines of the

initial cluster (i.e. there is no implementation of the sub-partition concept in current stable JBoss releases). The load balancing policies are (i) defined at deployment time, inside EJB deployment descriptors, (ii) integrated into client-side proxy code, and (iii) are non-adaptive (i.e. none of them considers the dynamic nature of the computational load of the clustered machines).

- As to the replication of EJB components (HA-EJB), there is (i) no replicated version of Message Driven Beans in stable JBoss releases, and (ii) no distributed locking mechanisms or distributed cache for the synchronization of Entity Beans. These beans can only be synchronized by using a row-level locking in the database. Thus, if an entity bean locks a database and fails, there is to be a mechanism, at the database level, responsible for unlocking the resources used by that bean.
- There is no unified management of the cluster (i.e. it is missing a cluster-wide configuration management, and it is only possible to connect directly to the JMX console of each node).

To conclude this Report we wish to summarize the current status of our work.

- The Interpreter Service, which parses an SLA in order to transform it into a Java object, has been currently implemented and integrated into the JBoss application server as an MBean.
- For the purposes of our discussion, we have assumed that the JBoss cluster of machines is based on a Local Area Network (LAN); however, in principle, this cluster can be geographically distributed across the Internet using, for example, VPN technology that abstracts out possible networking heterogeneities. Owing to this observation, we are currently exploring the possibility of constructing a VPN based network infrastructure that will allow us to distribute a JBoss cluster across at least five sites in Europe (namely, the sites of the TAPAS project partners) as if it was distributed on a local area network. One of the principal scopes of this exercise is to experiment the heterogeneous deployment discussed in the previous Subsection 3.2.
- In addition, in order to evaluate the feasibility of implementing our Configuration Service, we are assessing to what extent the JBoss application server can be programmed, so as to distribute the computational load dynamically, at run time. The testbed for this experiment will consist of a cluster of machines, running JBoss, which will be subjected to variable load conditions. Our plan is that this cluster of machines, initially based on the local area network in our Laboratory, will then be based on the VPN mentioned above.
- Finally, we wish to report that an initial implementation of the Configuration Service is running in our laboratory. Currently, this Service can automatically start up a JBoss node, and transfer nodes among partitions. This service runs as an Mbean, and is integrated in JBoss via JMX.

Configuration Implementations	Description	Advantages	Disadvantages
Heterogeneous Deployment	Different beans of the same application deployed in the machines of the cluster, both at deployment time and at run time.	Better utilization of resource availability (e.g. web components close to clients, EJB components related to database close to database servers)	Need to know all application component DDs. Distributed transactional manager and mechanisms to improve performances necessary for components migration. Adaptive load balancing policy missing: to be included
Homogeneous Deployment	Copies of the entire application deployed in the cluster (i.e. every machine with the same beans at deployment and at run-time)	Simpler solution than the previous. JBoss clustering used with all its features	Adaptive load balancing policy missing: to be included .
Homogeneous and Heterogeneous Deployment together	Copies of the entire application deployed in the cluster at deployment time. At run time some machines can have different set of beans of the same application	Simpler solution than the first one. Possibility to exploit the available resources at run-time. JBoss clustering used with all its basic features.	Distributed transactional manager to provide as well as mechanisms to manage migration activities. Adaptive load balancing policy missing: to be included

Table 1: Application deployment alternatives

5 References

[1] G.Ferrari, G.Lodi, F.Panzieri and S.K.Shrivastava "The TAPAS Architecture: QoS Enabled Application Servers", TAPAS deliverable D7, April 2003, Brussels.

[2] JBoss group "Feature Matrix: JBossClustering (Rabbit Hole)", 19th of March 2002.

[3] S.Labourey and B.Burke " JBoss Clustering 2nd Edition", 2002.

[4] http://www.javagroups.com/

[5] G.Ferrari and G.Lodi "Implementing the TAPAS Architecture", TAPAS Internal Draft, December 2003.

[6] S. Labourey "Load Balancing and Failover in the JBoss Application Server", 2001-2004 IEEE Task Force on Cluster Computing, http://www.clustercomputing.org

[7] J.Vuckovic et al. "JBoss Clustering Analysis", Bologna, February 2003.

[8] B.Burke and S.Lauborey "Clustering with JBoss 3.0", ONJava.com, October 2002.