TAPAS

IST-2001-34069

Trusted and QoS-Aware Provision of Application Services

QoS-aware Application Server: Design, Implementation, and Experimental Evaluation

(formerly entitled "Revised Container Interface Specification")

Report Version: D11

Report Delivery Date: 31 March, 2005

Classification: Public Circulation

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)

QoS-aware Application Server:
Design, Implementation, and Experimental Evaluation

Giorgia Lodi, Fabio Panzieri, Davide Rossi, Elisa Turrini

University of Bologna
Department of Computer Science
Mura A. Zamboni 7
I – 40127 Bologna
{lodig|panzieri|rossi|turrini}@cs.unibo.it

Table of contents

Appendix

1 Giorgia Lodi, Fabio Panzieri, "QoS-aware Clustering of Application Servers", Proc. 1st IEEE International Workshop on Quality of Service in Application Servers (QoSAS 2004), in conjunction with 23rd Symposium on Reliable Distributed Systems (SRDS 2004), Jurerê Beach Village, Santa Catarina Island, Brazil, 17 October 2004.

2 Davide Rossi, Elisa Turrini , "Testing J2EE clustering performance and what we found there", Proc. 1st IEEE International Workshop on Quality of Service in Application Servers (QoSAS 2004), in conjunction with 23rd Symposium on Reliable Distributed Systems (SRDS 2004), Jurerê Beach Village, Santa Catarina Island, Brazil, 17 October 2004.

3 Giovanna Ferrari, Santosh Shrivastava, Paul Ezhilchelvan, "An Approach to Adaptive Performance Tuning of Application Servers", Proc. 1st IEEE International Workshop on Quality of Service in Application Servers (QoSAS 2004), in conjunction with 23rd Symposium on Reliable Distributed Systems (SRDS 2004), Jurerê Beach Village, Santa Catarina Island, Brazil, 17 October 2004.

4 Paul Ezhilchelvan, Giovanna Ferrari, Mark Little, "Realistic and Tractable Modeling of Multi-tiered E-business Service Provisioning", Technical Report, The University of Newcastle upon Tyne, March 2005.

Abstract

In this Report we describe the design, implementation and experimental evaluation of a collection of services we have developed in order to enable application server technology to meet such Quality of Service (QoS) application requirements as service availability, timeliness and throughput. These services have been integrated within the JBoss application server, and their effectivess assessed via a priliminary experimental evaluation we have carried out. Further evaluations of our services are currently in progress; the results of these evaluations are expected to be available by 31 March 2005, end date of the TAPAS project.

1 Introduction

It is common industry practice to specify the QoS application requirements within a legally binding contract, termed Service Level Agreement (SLA). An SLA includes both the specification of the QoS guarantees an application hosting environment, such as an application server, has to provide its hosted applications with, and the metrics to assess the QoS delivered by that environment.

The definition of SLAs is a complex task, investigated in depth both in the literature and in the context of the TAPAS project; hence, we shall not discuss it further in this Report (the interested reader can refer to [2,7,24,28]). However, for the purposes of our current discussion, we wish to point out that we term *QoS-aware application hosting environment* (or QoS-aware application server) a hosting environment designed to honour the so-called *hosting* SLAs; i.e., the SLAs that bind that environment to the applications it hosts.

Current J2EE-based application server technologies (e.g., JBoss [13], JOnAS [16], WebLogic [3], WebSphere [11]) can meet only partially QoS requirements such as availability, timeliness, security, and trust of the applications they host, as these technologies are not fully instrumented for meeting those requirements (i.e., they are not designed to be *QoS-aware*).

In essence, in order to construct a QoS-aware hosting environment, the above mentioned SLAs are to be enforced, and monitored at run-time. This entails that possible deviations of the QoS delivered by the environment from that expected by the application must be detected, and corrective actions taken, before the SLA that binds the environment to the application is violated.

We propose that SLA enforcement and monitoring be carried out by two principal middleware services, namely a *Configuration Service* (CS) and a *Monitoring Service* (MS), which can be incorporated in the current application server technology.

In addition, as advanced application server technology such as JBoss, enables hosting of distributed, component-based applications within a cluster of application servers, the CS and MS mentioned above are to be designed so as to exercise SLA enforcement and monitoring over clustered application servers (hence, in the following, we use the phrase *QoS-aware clustering* to refer to a QoS-aware hosting environment constructed out of clustered, QoS-aware application servers).

Clustered application servers can typically be used in order to provide the applications with a highly available, fault tolerant, and scalable hosting environment. For both performance and fault tolerance purposes, load balancing is to be deployed within a cluster of application servers so as to distribute appropriately the computational load amongst those servers.

To the best of our knowledge, current open source, J2EE technology (e.g., JOnAS, JBoss) offers load balancing services which implement simple, non adaptive load distribution strategies. As discussed later, static load distribution may affect the QoS delivered by an application server cluster. In order to overcome this problem, we have developed a load balancing service that incorporates an adaptive load balancing strategy. This strategy can cope effectively with run time variations of both the clustered servers computational load, and the cluster configuration. Specifically, our load balancing service operates in conjunction with the CS and MS in order to implement QoS-aware application server clustering.

In this Report we introduce the design and implementation of the CS, MS, and load balancing service we have developed in order to build a QoS-aware application server, based on JBoss, which incorporates server clustering support. Specifically, we describe an extension of the JBoss clustering service we have developed in order to enable QoS-aware clustering of JBoss application servers. In addition, we discuss an initial experimental evaluation of an implementation of this JBoss extension we have carried out.

This Report is structured as follows. The next Section introduces the principal issues we have addressed in the design of our CS, MS, and Load Balancing Service. Section 3 summarizes their implementation. Section 4 describes our experimental results. Section 5 compares and contrasts our design and implementation with relevant related work. Section 6 provides some concluding remarks. Finally, the Appendix contains three papers, by members of the TAPAS project, introducing some of the concepts discussed in this deliverable Report, and a work in progress technical report discussing the modeling of the activities within an E-business site.

2 Design Issues

The principal issues we have addressed in the design of our QoS-aware application server include .(i) SLA enforcement and monitoring, (ii) application server structuring, and (iii) application deployment. In this Section we discuss these issues, in isolation; however, for the sake of completeness, we summarize first the overall TAPAS architecture, in order to position our discussion within the context of that architecture.

The Structuring of the TAPAS architecture [28] is illustrated in Figure 1, below. In this figure, the three subsystems depicted as patterned boxes are TAPAS specific components. As pointed out in [28], in the absence of these three subsystems, we have a standard application hosting environment, consisting of an application server constructed out of component middleware (e.g. J2EE).

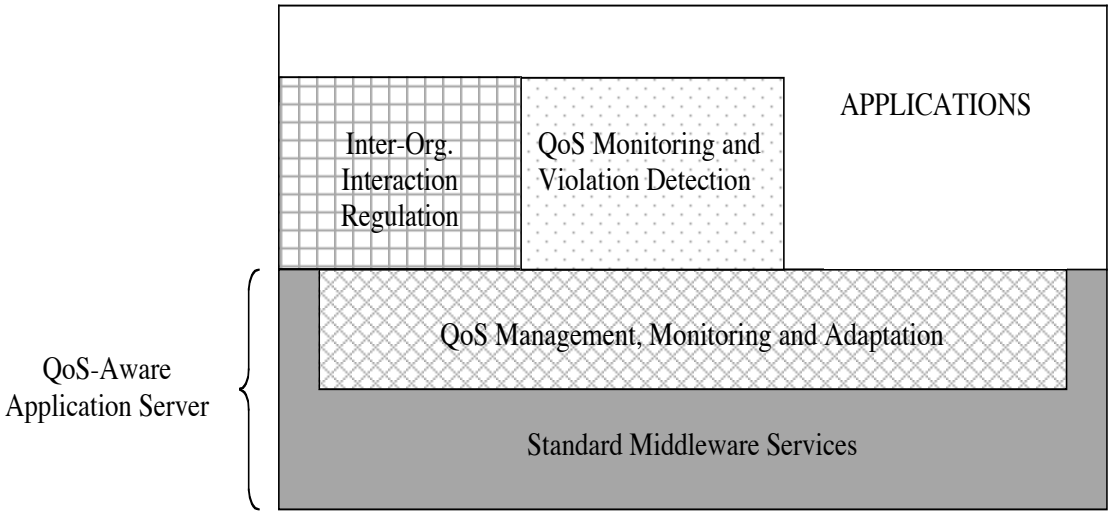

Figure 1: TAPAS Platform

In Figure 1, the *Inter-Organisation Interaction Regulation* subsystem polices all the cross-organisational interactions. This subsystem is responsible for managing electronic contracts (*x-contracts* [28]), represented using finite state machines and role based access control (RBAC) mechanisms for run-time monitoring and policing.

The *QoS Monitoring and Violation Detection* subsystem is used to ensure that an application meets its own QoS requirements, as stated in the SLA that application has with its hosting environment. Specifically, this subsystem monitors the external interactions of that application, at run time, so as to detect any violation of the application hosting SLA, and inform the interacting parties about that violation.

Finally, the *QoS Management, Monitoring and Adaptation* subsystem makes the underlying application server QoS-aware. In particular, this subsystem is responsible for the configuration and run time monitoring of the application server used to host a distributed application. In addition, this subsystem may *adapt* (i.e., reconfigure) the application server at run time, if the QoS delivered by this server deviates from that specified in the application hosting SLA.

Note that, within the TAPAS architecture, QoS monitoring occurs a two distinct levels of abstraction. Namely, it occurs at the application server level, in order to assess whether the application server resources deliver the required QoS, as derived from the hosting SLA (see next Subsection); in addition, it occurs at the higher level, in order to assess whether the hosting environment as a whole meets the hosting SLA.

In the following, we shall focus on the QoS-aware Application Server, only; i.e., the *QoS Management, Monitoring and Adaptation* subsystem of Figure 1. Thus, for the purpose of this Report, we will not discuss any further both the Inter-Organisation Interaction Regulation, and the QoS Monitoring and Violation Detection subsystems. In addition, we shall assume that an Application Service Provider (ASP) provides 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 an application server, such

as JBoss. The customer application QoS requirements are specified within a hosting SLA.

2.1 SLA Enforcement and Monitoring

Within the above scenario, it is necessary to construct, and maintain at run time, a hosting environment in which the customer applications can be deployed and run, and their QoS requirements reliably met (i.e., their hosting SLA honoured). To this end, the *QoS Management, Monitoring and Adaptation* subsystem of Figure 1 incorporates the CS and the MS mentioned earlier. In addition, in order to enable effective clustering of application servers, this subsystem embodies a Load Balancing Service which operates in conjunction with these CS and MS.

The CS is responsible for configuring an application hosting environment (be this a single application server, or a cluster of servers) so that it meets effectively the hosting SLA of a customer application. Thus, in essence, the CS takes in input a customer application hosting SLA, and discovers the available system resources that can honour that SLA. Provided that the discovered resources be sufficient to meet that input SLA, the CS reserves those resources, generates a "resource plan" (see below) for the hosted application, and sets up the QoS-aware application hosting environment for that application. In case the CS discovers that there are not sufficient resources to host that application, it returns an exception. (Typically, one such an exception can be handled either by rejecting the application hosting request, or by offering a reduced service, for example, depending on the policy implemented by the Application Service Provider owning the hosting environment.)

In essence, the resource plan mentioned above specifies the QoS levels the application expects from its hosting environment. These QoS levels are derived from the application hosting SLA. Specifically, the resource plan includes an interval of QoS values within which degradations of the QoS delivered by the hosting environment can be tolerated, before the hosting SLA be violated. This interval ranges between the so-called *warning* and a *breaching* points.

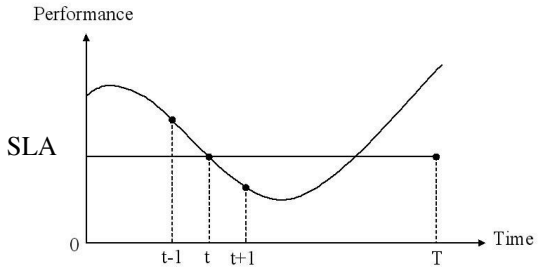

Figure 2: Warning and the breaching points

In general, a *breaching* point is a value associated to a QoS parameter (e.g. throughput, response time) of a given resource in a hosting environment. If the QoS delivered by that resource reaches that value, the SLA(s) binding the resource home environment to its hosted application(s) can be violated; a *warning* point is a QoS value indicating the limit up to which a degradation of the QoS delivered by that resource can be tolerated "safely" (i.e., without causing SLA violations).

The performance warning and a breaching points of a specific resource in a hosting environment are depicted in Figure 2, above. As illustrated in this Figure, if the performance delivered by that resource reaches its warning point, it is necessary that corrective actions be carried out (e.g. some adaptation strategy can be invoked), in order to avoid SLA violations. If these actions fail, the resource may reach its breaching point, thus violating the SLA. In this latter case, an exception is to be raised that can be dealt with at some higher level of abstraction than that of the hosting environment (e.g. at the application level).

Note that the breaching points included in the resource plan can be derived from the SLA. In contrast, the warning points (or thresholds, in the following) can be determined either by means of application benchmarking, carried out prior to application deployment, for example, or by means of techniques based on the modelling and simulation of the hosting environment under different load or failure conditions. (These techniques are outside the scope of this Report; hence, in the following, we assume that warning thresholds are defined by means of application benchmarking.)

In our implementation of the CS, the resource plan is included in an *AgreedQoS* object, generated by the CS, and used as input to the MS. The MS is responsible for monitoring the hosting environment at application run time, so as to detect possible violations of the SLA. In order to prevent those violations, the MS is designed so that it takes appropriate actions if it discovers that a resource reaches its warning point. Thus, for example, the MS can make use of a predefined "overload" warning point in order to detect dangerous load conditions that may lead to server overloading. In case that warning point is reached, the MS invokes the CS, and requires that the application hosting environment be reconfigured appropriately, so that it can adapt to the new load conditions, and continue to honour the application SLA. Incidentally, it is worth observing that, in practice, the MS may have to make use of a number of warning points (e.g., a throughput warning point, a response time warning point), which contribute to define one such warning point as the overload one mentioned above.

Finally, in addition to the CS and the MS, SLA enforcement in an application server cluster requires the use of a load balancing service that distributes appropriately the client requests load amongst the clustered servers. Typically, this service can contribute to honouring the hosting SLA by both preventing the occurrence of server overload in each clustered application server, and avoiding the use of resources which have become unavailable (e.g., failed) at run time.

2.2 QoS Monitoring and Adaptation Services

Typically, the CS and the MS exercise their activity over both the internal resources of each application server instance in the application hosting environment, and the set of clustered server instances that form this environment; i.e., they are responsible for configuring and monitoring both a single application server and a cluster of servers.

Owing to this observation, the CS and the MS can be conveniently thought of as operating at two distinct levels of abstraction, that we term the *micro-resource* and the *macro-resource* levels, respectively. The former level consists of resources, such as server queues, thread and connection pools, internal to each individual application

server; the latter level consists of such resources as the group of clustered application servers, and their IP addresses.

Thus, for example, the CS at the micro-resource level is responsible for sizing appropriately an application server request queue, in order to enable that server to deal with an (anticipated) maximum number of concurrent requests, and to maintain its responsiveness. In contrast, in order to meet possible load balancing and responsiveness requirements, the CS at the macro-resource level may have to modify the cluster configuration at application run time, e.g., by enabling one (or more) new application server instance(s), or by replacing a crashed application server instance with an operational one.

The MS at the micro-resource level monitors the QoS (e.g., throughput, response time) delivered by the single application server, and requires the *server reconfiguration* when the delivered QoS reaches a predefined warning threshold. At the macro-resource level, instead, the MS monitors the QoS delivered by the clustered application servers, and requires *cluster reconfiguration* in case the cluster delivered QoS reaches a predefined warning threshold.

The interface between the micro and the macro resource levels can be thought of as consisting of primitive operations and data objects that enable the macro-resource level both to obtain micro-resource QoS data (e.g., server throughput, server response time, JVM free memory) from the micro-resource level, and to provide this level with QoS requirements derived from the SLA.

In the next two subsections we describe in detail the design of the CS and MS at the macro-resource level. (The micro-level CS and MS are introduced in [9.a], included in the Appendix of this Report.) Following our description of the macro-resource level CS and MS design, we introduce, in a separate subsection, both the design of the load balancing service we have developed as part of the QoS Management, Monitoring and Adaptation Subsystem of Figure 1, and our adaptive load balancing strategy.

2.2.1 Configuration Service

The principal responsibilities of the CS consists of configuring the application server cluster at SLA deployment time, and possibly reconfiguring the cluster at run time, if the QoS the cluster delivers deviates from that specified in the hosting SLA. The skeleton code in Figure 3 illustrates the operation of the CS.

As shown in this Figure, the CS is created and started at application server start up time. The actual cluster configuration initiates at SLA deployment time (i.e., when an SLA is effectively deployed in a clustered application server). Typically, the CS instance of the application server where an SLA deployment occurs becomes the cluster configuration leader. This CS instance takes as input the SLA, and parses it in order to extract from it the relevant QoS parameters that will guide and determine the required cluster configuration.

In constructing the cluster configuration, the CS produces two objects which are essential to the cluster operation; namely, these objects are the *AgreedQoS* object and the *LoadBalancerFactor* object, described below.

The *AgreedQoS* object contains essentially the resource plan mentioned in Section 2.1. It includes both the *warning* and the *breaching* points of the clustered resources.

```
// START  UP  TIME
        create-service();
        start-service();
// SLA  DEPLOYMENT   TIME
        enableConfiguration(SLA) {
           // Elect cluster leader
           leaderElection(SLA);
           // Configure cluster
           clusterConfiguration(SLA, getMembership);
           // set configuration state of cluster members
           setClusterState();
        }
        clusterConfiguration(SLA, membership) {
                // Reconfigure cluster if n. of members not sufficient to meet SLA
                if (membership < numberMemberRequiredForSLA) {
                        addNewInstancesReconfiguration();
                }
                // Get microDataQoS from every cluster member
                getMicroDataQoS();
                //Compute Load Balancer Factor and create Agreed QoS
                computeLbFactorAndAgreedQoS(microDataQoS, membership);
        }
        computeLbFactorAndAgreedQoS(microDataQoS, membership) {
                // Compute Load Balancer Factor
                computeLoadBalancerFactor(microDataQoS, membership);
                // Create agreed QoS
                createAgreedQoS(microDataQoS, membership);
        }
        getMicroDataQoS() {
                  for (each member of the cluster) {
                  microDataQoS = getAvailability();
        }
// RUN  TIME
    addNewInstancesReconfiguration() {
        }
    rearrangeAgreedQoS() {
        // get micro data from cluster members
        getMicroDataQoS();
        //Compute Load Balancer Factor and create Agreed QoS
        computeLbFactorAndAgreedQoS (microDataQoS, membership);
        // set (re)configuration  state of cluster members
        setClusterState();
    }
```

Figure 3: CS skeleton code

As the resources of interest for our macro-level CS are application server instances, in our current design the warning and breaching points associated to each of these resources are relative to the following three QoS parameters, only:

- *ResponseTime*: the time elapsed between the delivery of a client request to an application server, and the transmission of the reply to that request from that server;
- *Throughput*: the number of client requests that can be served by the application server within a specific time interval;

- *FreeMemory*: the size of the free memory in (the JVM of) each application server.

It is worth observing that the breaching points for these three parameters can be derived from the requirements included in the hosting SLA. In contrast, the warning points are to be calculated based on, for example, the expected load, or even specific ASP policy decisions concerning the risks the ASP wishes to assume as to the violation of a hosting SLA.

In our design, we have not addressed issues of warning points assessment, as these issues fall outside the scope of our current work. Rather, we have assumed that the warning points are set to values which are sufficiently close to their relative breaching points to enable possible reconfiguration of the hosting environment, before those breaching points are reached. Thus, for example, assume that the hosting SLA of a Web application specifies that the response time of each client request to that application must not exceed 6s, and 1 s is the time required to reconfigure the application hosting environment. In this circumstances, the response time warning point can be set to a value which is 20% lower than 6s (i.e., it can be set to 4.8s) so as to make sure that, if the warning point is reached, the hosting environment reconfiguration can be carried out safely, before the SLA is violated.

Hence, in general, when a warning point is reached at run time, some adaptation strategy can be invoked that reconfigures the hosting environment so as to reduce the risk of violating the hosting SLA (i.e., reaching a breaching point). As already mentioned, adaptation may fail and a breaching point may be reached, eventually, causing that an exception be raised.

In our current implementation, adaptation is firstly applied at the micro-resource level, by adjusting single application server parameters (e.g., connection pool, thread pool), and secondly at the macro-resource level, if the micro-resource level adaptation turns out not to be sufficient for the required reconfiguration purposes.

Specifically, the macro-resource level reconfiguration consists of the following two principal activities: (i) adding new nodes to the cluster, and (ii) maintaining a so-called *availability index* associated to each clustered application server (see below). The activity (i) can be necessary both in case a cluster is to be augmented with additional resources (e.g., in order to cope with dynamically increasing load conditions), and in case a clustered server fails and is to be replaced by one (or more) server(s). In both these cases, it may be convenient to maintain a pool of spare servers available for inclusion in the cluster, at any time, so as to eliminate the overhead induced by bootstrapping a new application server.

As to the activity (ii), we wish to mention that the *availability index* is a positive value associated to each clustered application server which essentially indicates the percentage of usage of each server instance at a given time. This index is determined by te CS on the basis of the response time, the throughput, and the available memory at the server instance itself, and is used by the Load Balancing Service in order to distribute the load among the clustered servers, when an adaptive load balancing strategy is deployed (see Subsection 2.2.3).

2.2.2 Monitoring Service

The skeleton code of the macro-resource level MS is depicted in Figure 4, below. The MS consists of the following five principal components.

1) The *Membership Interceptor* – This interceptor is used to retrieve information about the cluster membership (e.g., cluster dead members, current cluster members, new cluster members).

2) The *Client Request Interceptor* – This interceptor is used to intercept the client requests in order to evaluate the cluster performance for specific application requests; the cluster performance consists of the set of throughput and response time values that allow one to detect whether or not the nodes of the cluster are overloaded. This interceptor is used in the Load Balancing architecture described in the next Subsection, and is named *Request Interceptor*.

3) The *Measurement Service* – This service is used in order to store periodically monitoring information (for logging purposes).

4) The *Cluster Performance Monitor* – This component cooperates with the Evaluation and Violation Detection Service in order to assess the actual response time and the throughput delivered by the application server cluster, and to detect possible SLA violations. Specifically, the Cluster Performance Monitor (i) obtains the data required to assess response time end throughput from the client request interceptor (e.g., number of received requests, for specific application operations, number of served requests, for specific application operations), (ii) computes the average request response time and throughput, and (iii) sends the results of its computations to the Evaluation and Violation Detection Service. This latter service is invoked in order to check the adherence of those values to the SLA requirements.

5) The *Evaluation and Violation Detection Service* – This service is responsible for checking whether the QoS delivered by the clustered application servers meet the hosting SLA requirements. Specifically, this component detects, at run time, variations in the operational conditions of the clustered servers, which may affect the QoS delivered by the cluster, and triggers the cluster reconfiguration, if necessary.

2.2.3 Load Balancing Service

In the design of our Load Balancing Service for clustered application servers, we have evaluated both a "request-based" (or "per-request") load balancing approach, and a "session-based" (or "per-session") load balancing approach.

In "request-based" load balancing, each individual client request is intercepted by the Load Balancing Service, and dispatched to an application server for processing, according to some specific load distribution policy (see below). Thus, two consecutive requests from the same client may be dispatched to two different servers.

In contrast, using "session-based" load balancing, a specific client session (i.e., a sequence of client requests) is created in one of the clustered application server, at the time a client program requires access to the application hosted by that server; every future request from that client will be processed by that server (these client-server sessions are termed "sticky sessions"). Thus, the Load Balancing Service intercepts

each client request and, depending on the sticky session the request belongs to, dispatches it to the appropriate server.

```
//START UP TIME SLA DEPLOYMENT TIME AND RUN-TIME
start-monitoring() {
	while(true) { // Get cluster membership by using the membership interceptor
		getMembership();
		if (membershipChanged) {
		getNewMembers();
		getDeadMembers();
		// Send log to MeasurementService and save it in stable storage
		saveLogsWithTheMeasuramentService();}}}
	getDeadMembers() {
		if (group_leader_dead) { /* Elect new leader */ }
		// SLA evaluation
		evaluationViolationDetectionService.SLAEvaluation(members); }
	getNewMembers() { /* Call rearrangeAgreedQoS to include membership changes */}

//RUN TIME
		// Get client requests via client request interceptor
		getClientRequests();
		// Invoke Cluster Performance Monitor to compute RT and Throughput
		ClusterPerformanceMonitor.computeRespTime();
		ClusterPerformanceMonitor.computeThroughput();
	class ClusterPerformanceMonitor {
		// Record number of arrived requests
		// Record number of served requests
		// Compute response time
		// Compute throughput
		// Evaluate response time via EvaluationViolationDetection Service
		evaluationViolationDetectionService.SLAEvaluationRT(computedRT);
		// Evaluate Throughput via EvaluationViolationDetection Service
		EvaluationViolationDetectionService.SLAThroughput(computedTh);
	}
	class EvaluationViolationDetectionService {
		SLAEvaluation(membership) {
			if (membership.size < membershipRequiredSLA) {
				//invoke CS addNewInstancesReconfiguration
			} else {
				//No need to add new replica but
				//invoke CS rearrangeAgreedQoS
			}
		}
		SLAEvaluationRT(computedRT) {
			getRTThresholds();
			if (computedRT > SLART) {
			   //SLA VIOLATED; return exception
			else {
			  if (computedRT > WPRT)
			  // RespTime warning point reached; invoke adaptation
			}
		}
		SLAEvaluationTh(computedTh) {
			getThThresholds();
			if (computedTh < SLATh) { //SLA VIOLATED
			} else {
				if (computedTh > WPTH)
			  // Throughput warning point reached; invoke adaptation
			}
		}
	}
```

Figure 4: MS skeleton code

The relative advantages, and shortcomings, of the two approaches above can be summarised as follows. In principle, the "request-based" load balancing allows one to construct a hosting environment that can support highly available applications. Provided that the application state be maintained mutually consistent across the clustered application servers, the crash of a server can be masked by routing the client requests to a different server in the same cluster, transparently to the client program. In practice, the cost of maintaining consistency among clustered servers can be very high, in terms of both performance and memory overheads caused by the consistency algorithms (typically, state replication among the clustered servers can be required, as anyone of these servers can be used, without distinction, to serve incoming client requests at any time) [34].

In contrast, "session-based" load balancing trades availability for performance and scalability. This approach can be implemented so as to impose no consistency requirements among the clustered servers, as all the client requests belonging to the same session are served by the same server. However, if a server crashes while it is serving a client session, that session cannot be recovered, at the application server level. Thus, either the client program implements its own recovery mechanisms, or this program may have to start its session again (at the risk of causing multiple executions of earlier requests). It is worth mentioning that session replication across multiple replica servers may well be implemented at the application server level, and used to provide clients with transparent fault tolerant support to server crashes. However, the cost of maintaining mutually consistent client session replicas across multiple servers may make this solution as impractical as that mentioned above, in the case of "request-based" load balancing.

In view of the above observations, the architecture of the Load Balancing Service we have developed includes support for both request-based load balancing, and sticky session-based load balancing; this architecture is depicted in Figure 5 below, and summarized in the following.

Our Load Balancing Service can be thought of as a reverse proxy server which interfaces the clients of an application to the clustered application servers (nodes) hosting that application. This Load Balancing Service can accommodate a number of load balancing policies; typically, this Service requires to be configured at cluster configuration time so as to use, at run time, one specific load balancing policy out of those it incorporates.

The architecture of our Load Balancing Service consists of the following four principal components: namely, the Request Interceptor, the Load Balancing Scheduler (LBScheduler), the HTTP Request Manager, and the Sticky Session Manager, illustrated in Figure 5. Note that LBScheduler component in Figure 5 embodies four load balancing policies. Only one of these policies can be selected at cluster configuration time; that policy will be used at run time for load balancing purposes. It may be worth mentioning that the load balancing policies shown in Figure 5 are those actually incorporated in our current Load Balancing Service implementation; however, our Service is structured so that the set of policies embodied by the LBScheduler can be extended easily with additional load balancing policies.

As mentioned earlier, our Load Balancing Service can balance the client request load either on a per-request basis or on a per-session basis. Yet again, the decision as

to which "granularity" (i.e., either per-request, or per-session) of load balancing to use is to be taken at cluster configuration time.

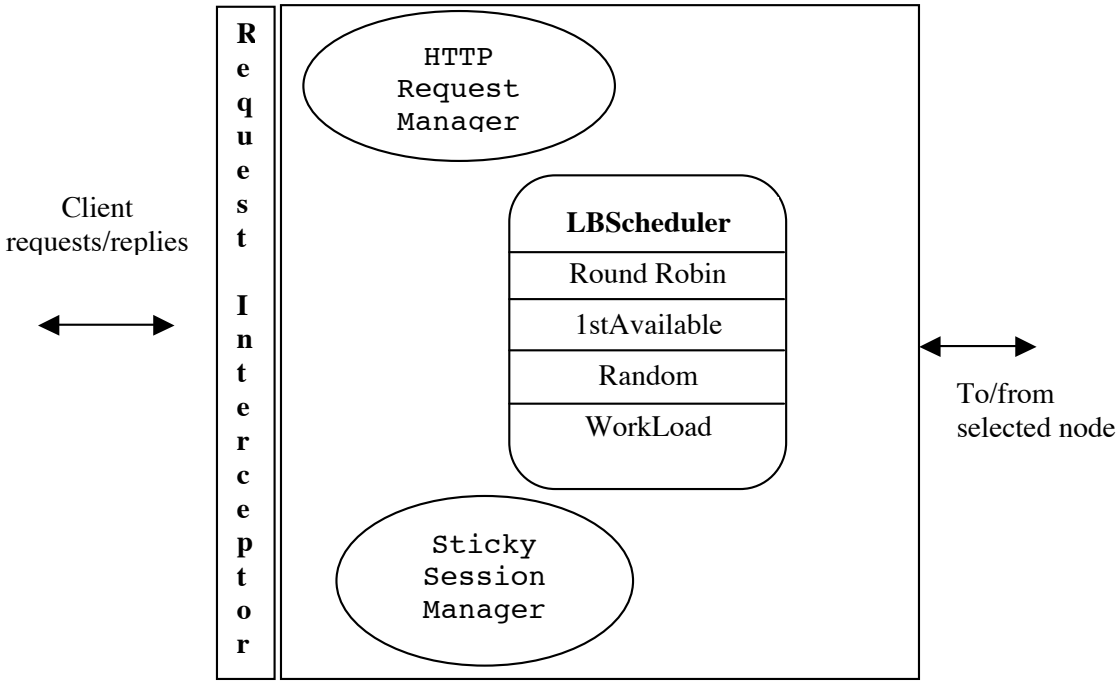

Figure 5: Load Balancing Service Architecture

If per-request load balancing has been enabled, the Request Interceptor intercepts each client request, and invokes the HTTP Request Manager. This Manager firstly interrogates the LBScheduler in order to obtain the address of a target node that can serve that request. The LBScheduler returns the requested address, based on the specific load balancing policy it has been configured to use. As the HTTP Request Manager obtains the target node address, it manipulates the client request appropriately in order to forward it to this target node, and to enable it to return the reply to the Load Balancing Service itself. As the Load Balancing Service receives a response to a client request, it forwards it to that client. The next request from the same client will be dealt with the same way; hence, there will be no guarantee that the same target node will be selected for serving it.

In contrast, if the Load balancing Service has been configured to use sticky sessions, the Sticky Session Manager cooperates with the LBScheduler to identify a target node that will serve the client requests for the entire session. Thus, the Sticky Session Manager maintains the various, active client-server sessions for the entire duration of these sessions.

A few observations concerning the design of our Load Balancing Service are in order. Firstly, we wish to mention that indeed a Load Balancing Service architecture based on a reverse proxy may show some limitations, in terms of scalability, and robustness, for example. However, one such an architecture has the merit of enabling the development of a Load Balancing Service which is fully transparent to both the client and the clustered application servers. Hence, on the balance, we have favoured the reverse proxy based architecture introduced above.

Secondly, it is worth observing that our Load Balancing Service has been designed so as to be completely independent of the policy used to select the target node. In fact, the Service architecture we have developed allows developers to plug into it the load balancing policies of their choice.

Finally, we wish to point out that the "Random", "Round Robin", and "1st Available" load balancing policies can be found in existing implementations of open-source J2EE platforms. However, these policies are all non-adaptive, i.e., they are unable to adapt to variations of the QoS delivered by the clustered application servers. As we are interested in the evaluation of adaptive load balancing policies, we have developed the so-called WorkLoad policy, described below.

This policy enables the LBScheduler to select the most lightly loaded server among the clustered servers, in order to serve either a request or a client session. Specifically, the LBScheduler uses the earlier mentioned *availability index*, generated by the CS and associated to each clustered application sever, and selects the clustered server with the highest *index* value. For each application server, this *index* is constructed out of the three QoS parameters of interest, mentioned earlier: namely, the server response time, the server throughput, and the server free memory. The CS obtains the value of each of these parameters from the micro-level resource manager at each node, and computes the availability index of each node as follows.

Let *N* be the number of clustered application servers. At cluster configuration time, the CS assesses, for each server *i*, the percentage of the total computational load that *i* is supporting, in terms of *i*'s response time, throughput, and free memory (below termed RespTime, Throughput, and Mem, respectively). To this end, the CS calculates the following three Factors:

1) the $\text{ResponseTimeFactor}_i = ((1/\text{RespTime}_i)*100)/\text{TotalRespTime}$, where: $\text{TotalRespTime} = \Sigma_{i=1}^{N} \text{RespTime}_i$,
2) the $\text{ThroughputFactor}_i = (\text{Throughput}_i *100)/\text{TotalThroughput}$, where: $\text{TotalThroughput} = \Sigma_{i=1}^{N} \text{Throughput}_i$, and
3) the $\text{MemoryFactor}_i = (\text{Mem}_i *100)/\text{TotalMem}$, where: $\text{TotalMem} = \Sigma_{i=1}^{N} \text{Mem}_i$.

Thus, in essence, the higher the response time of a server *i*, the lower its $\text{ResponseTimeFactor}_i$. In contrast, the larger the throughput (or the available memory) of a server *i*, the larger the percentage of the total cluster throughput that server delivers (or the percentage of the total available memory made available by that server in the cluster).

The availability index can be obtained as the median of the three Factors above, assuming that each QoS parameter has the same weight in the construction of this index.

$\text{AvailabilityIndex}_i =$

$(\text{RespTimeFactor}_i + \text{ThroughputFactor}_i + \text{MemFactor}_i) / \text{NumOfQoSParamenters}$,

where: $\text{NumOfQoSParamenters} = 3$

In conclusion, it is worth observing that the availability index is initialised to a default value in each application server, and dynamically adjusted at cluster configuration and reconfiguration time. Hence, our WorkLoad policy enables adaptation of the hosting

environment as variations of the QoS delivered by this environment occur, causing cluster reconfiguration.

2.3 Application Deployment

Issues of application deployment and application components replication in a clustered hosting environment can play a relevant role in the implementation of our Configuration Service. In this Subsection, we introduce these two issues, and examine three alternative implementations of our Service.

To begin with, it may be worth recalling that *homogeneous* application deployment entails that each node in a cluster of nodes 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, the use of this latter form of deployment is not recommended with current application server technology, such as JBoss, as that technology leaves as yet unsolved a number of problems related to the heterogeneous deployment, including lack of distributed locking mechanisms for use from entity beans, and cluster-wide configuration management.

However, we believe that heterogeneous deployment, in contrast with the homogeneous one, can be particularly attractive when applied to component-based applications, as it enables the distribution of the application components across a cluster of machines, in a controlled manner.

Typically, component based applications 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. In this context, the clustered application servers can be specialized; thus, for example, one application server instance can be configured for optimum performance for the support of transactions and Entity Beans, whereas another application server instance can be configured for optimum performance in the support of Session Beans.

In addition, if one assumes a rather conventional 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 as possible to the clustered database servers. This can be done both at deployment time, when the CS 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 MS).

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. 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 [4]:

- 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 CS to distribute the computational load so as to optimise the use of the available resources in the cluster. In contrast, its principal shortcoming is represented by its inherent complexity. Typically, this approach requires that the CS know all the application component DDs, in order to distribute those components. In particular, these DDs need to be carefully examined by the CS in order to maintain the local interfaces. 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).

HOAD – If support for homogeneous deployment is required (i.e. copies of the entire application are located in every node of the cluster), the CS has to establish at SLA deployment time the most suitable cluster that can host the application. In this case, the entire application can be deployed in only one node of the cluster; then, the JBoss Farming service implements its distributed deployment. If the hosting environment conditions change at run time, and the SLA is about to being violated, the CS has to either choose a different cluster, or create a new one, and reorganize the application deployment.

The implementation of this solution appears to be simpler than the HEAD solution discussed above, as it allows one to use the current JBoss clustering framework. However, in order to be effective, a HOAD solution requires that an adaptive load balancing policy, which enables the clustered environment to meet the hosting SLA, be incorporated in the clustering mechanism. (Note that the load balancing policies currently available in the JBoss application server are static, and suffer of a number of limitations that are introduced in Section 3, below.)

HHAD – Finally, the two approaches above can be combined as follows. At deployment time, the CS can execute homogeneous deployment by replicating component classes using the JBoss Farming service; thus, it will create a cluster of application servers 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 CS can decide whether to (i) add more replicas of the entire application, (ii) add a certain number of machines to the initial cluster, i.e. to augment the number of available resources, (iii) migrate application components from overloaded machines either to other ones in the current cluster, or different machines that do not belong to this cluster.

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 to (i) apply an adaptive load balancing policy, (ii) provide a distributed transactional manager, and (iii) both manage migration operations, and improve the performance of those operations.

To conclude this Subsection, we wish to point out that we have decided to adopt the HOAD approach in view of its lower complexity, and to introduce a new adaptive load balancing policy in our QoS-aware application server, in order to overcome the limitations of the available static load balancing mentioned above (and discussed in the next Section).

3 Implementation

The CS, MS, and Load Balancing Services introduced earlier have been implemented as an extension of the JBoss application server. In particular, a so-called TAPASCluster JBoss server configuration has been created, which incorporates our three Services above.

In summary, this configuration enables the clustering of JBoss application servers in a cluster of machines, as illustrated in Figure 6. Clients, typically browsers, may use distributed applications, homogeneously hosted by the clustered servers, by issuing HTTP requests to those applications. These requests are effectively intercepted by a one of the clustered servers (elected at cluster configuration time as the cluster leader), and dispatched to the server instance with highest availability index, via the load balancing service.

In the following, we shall describe in detail our implementation. However, for the sake of completeness we introduce first the JBoss application server and its clustering service. (The reader familiar with this technology can skip the next two Subsections.)

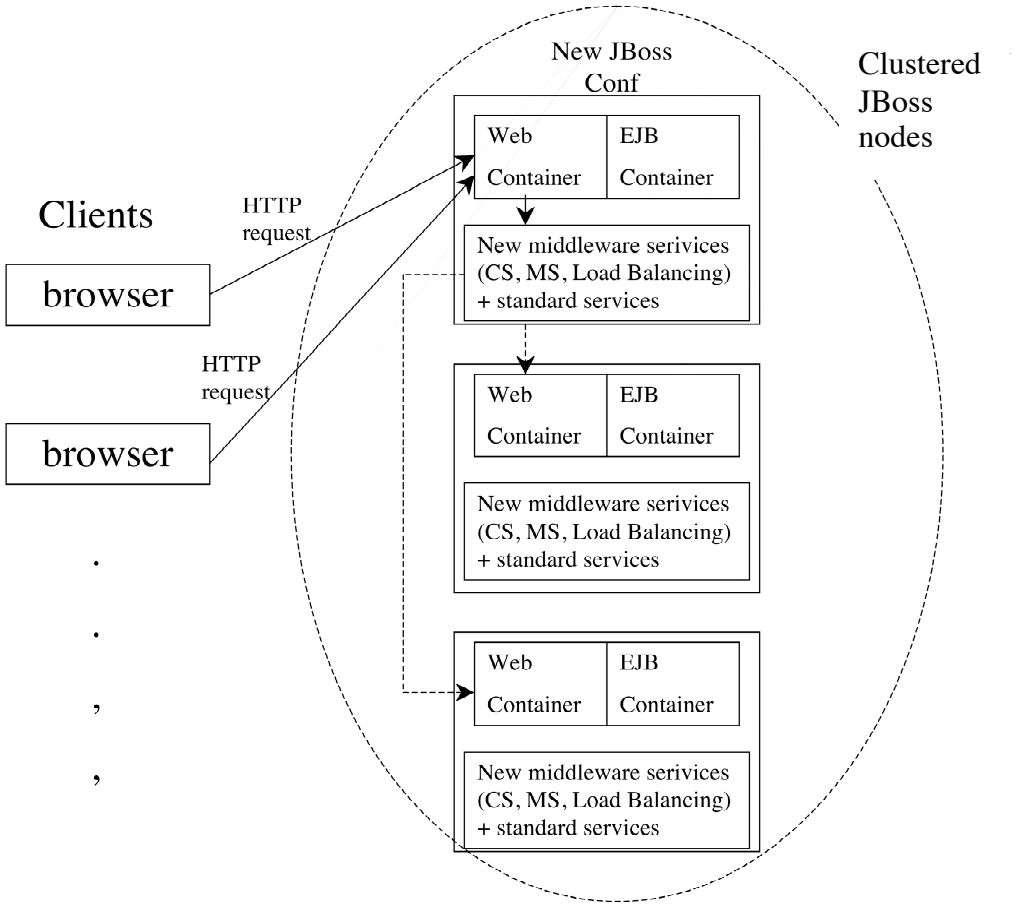

Figure 6: The TAPAS platform

3.1 The JBoss application server

JBoss consists of a collection of middleware services for communication, persistence, transactions and security. These services interoperate by means of a microkernel termed Java Management eXtension (JMX) [29]. Specifically, JMX provides Java developers with a common software bus that allows them to integrate components such as modules, containers and plug-ins. These components are declared as Managed Beans (MBeans) services, which can be loaded into JBoss, and can be administered by the JMX software bus. The MBeans are the implementation of all the manageable resources in the JBoss server; they are represented by Java objects that expose interfaces consisting of methods to be used for invoking the MBeans.

As mentioned earlier, a number of JBoss applications servers can be clustered in a network. A JBoss cluster (or *partition*) consists of a set of *nodes*. A node, in a JBoss cluster, is a JBoss application server instance. There can be different partitions on the same network; each cluster is identified by an individual name. A node may belong to one or more clusters (i.e., clusters may overlap); moreover, clusters may be further divided into *sub-clusters*. Clusters 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. (However, heterogeneous deployment is not encouraged for the reasons introduced earlier.)

The JBoss Clustering service [23] is based on the framework depicted in Figure 7 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 [15], 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.

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).

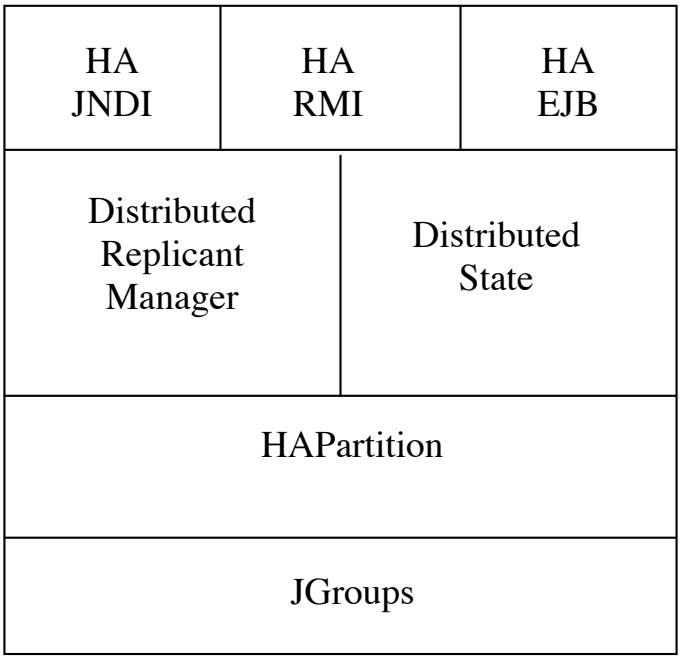

Figure 7: JBoss Clustering Framework

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.

3.1.1 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 [22], depicted in Figure 8 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).

JBoss adopts the model 3) in Figure 8, 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.

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 [22]. The de-typed invocation is passed through a set of client-side interceptors, as depicted in Figure 9, below. The load balancing and failover mechanisms are located in the last interceptor of the chain (i.e., the interceptor C in Figure 9).

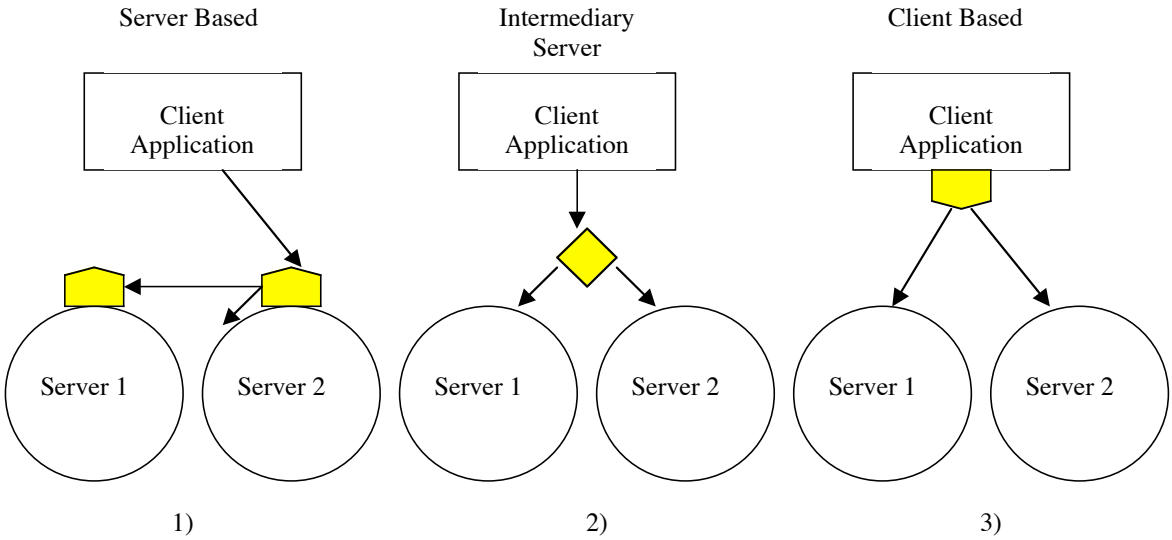

Figure 8: Clustering implementation models

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

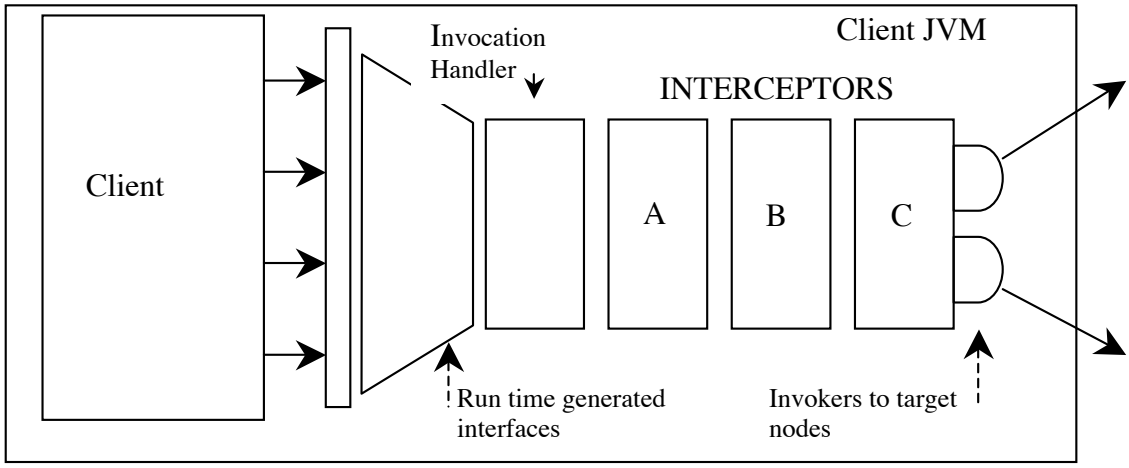

Fig. 9: Client-side interceptors

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.

The above load balancing policies 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 10, below.

Note that the 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, illustrated in Figure 9. Thus, the load balancing in JBoss can be augmented with user-defined, adaptive strategies that select the target machines at run time, based on the actual computational load of those machines.

It is worth observing that the above strategies are not used in our approach to load balancing as: (i) they operate at the RMI level; in contrast, our load balancing operates at the HTTP request level, and (ii) in our architecture we have decided to maintain the Web and EJB containers co-located in the same JVM, in order to exploit the JBoss local RMI optimization (hence, in this context, RMI load balancing is never used).

```
<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 10: Deploying load balancing policies in JBoss

Finally, note that the latest JBoss version incorporates a so-called HTTPLoadBalancer Service which implements a response time based adaptive load balancing strategy, for HTTP sessions only. This strategy is implemented at a higher level of abstraction than the RMI level mentioned above, and operates regardless of any hosting SLA; hence, we decided not to use it, and to create a new JBoss server configuration, instead (i.e., the TAPASCluster configuration mentioned earlier). This configuration provides the hosted applications with the QoS-aware clustering services described earlier in this Report; namely, the macro-resource level CS, MS, and Load Balancing Services. The Load Balancing Service incorporates the adaptive load balancing strategy introduced in Subsection 2.2.3. The implementation of these three services is described in detail in the next two Subsections.

3.2 The TAPAS MacroResourceManager Implementation

The macro-resource level CS and MS are implemented by an MBean we have termed MacroResourceManager, as illustrated in Figure 11.

The MacroResourceManager uses the following two MBeans: the MeasurementService MBean, which saves periodically the cluster state, and the SLADeployer MBean, which transforms the input SLA, specified in a XML form, into a Java object.

The MacroResourceManager implements the MS based on the monitoring architecture described in [24]. As illustrated in Figure 12, this implementation uses the above mentioned MeasurementService MBean, and two specific classes, termed Evaluation and Violation Detection Service, and Macro Resource Monitoring, respectively.

The Evaluation and Violation Detection Service is responsible for monitoring, at run time, the adherence of the run time execution environment to the SLA; i.e., it detects whether the QoS delivered by the run time environment (as obtained from the Measurement Service) is close to violating the SLA, and decides to start the cluster reconfiguration, if necessary.

The Macro Resource Monitoring is enabled by the MacroResourceManager, which starts the monitoring thread. This thread detects i) the current view of the cluster membership, ii) new members that join the cluster, and iii) dead members that leave the cluster. In order to carry out its task, the Macro Resource Monitoring uses the JGroups communication interface available in JBoss, and the JBoss clustering framework.

Finally, the current implementation of the Macro Resource Monitoring sends periodically the data about the cluster membership, obtained from the JGroups framework, to the Measurement Service. This latter Service maintains these data in stable storage for logging purposes.

The CS in the MacroResourceManager MBean implements a distributed cluster configuration protocol, which can be summarized as follows. Assume that a JBoss cluster is set up, and that homogeneous application deployment is to be carried out within that cluster. Each JBoss node in that cluster embodies a CS instance in its own MacroResourceManager MBean; this MBean is identified by a cluster-wide unique identifier (ID), assigned by the JGroups view management protocol.

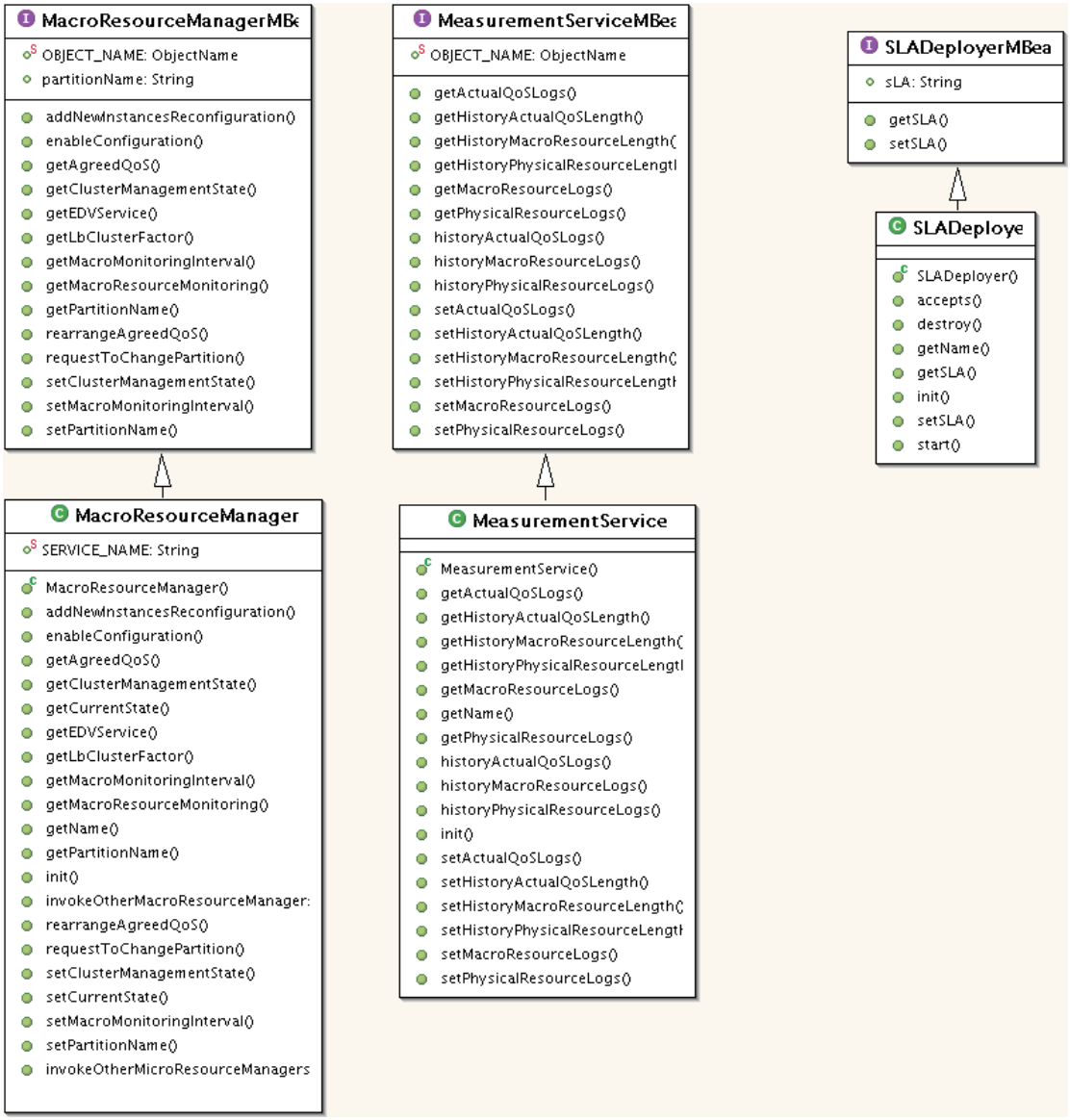

Figure 11: UML diagram of the MBeans of the macro-resource level

In order to configure the application hosting environment, the actual application deployment is preceded by what we term an SLA deployment phase. In this phase, an SLADeployer is provided with an application SLA. This SLADeployer enables its local MacroResourceManager, which becomes the MacroResouceManager Leader of the cluster configuration. This Leader examines the input SLA, and contacts its peer MacroResourceManagers in the cluster in order to i) discover the resource availability at these MacroResourceManager nodes, and ii) construct a suitable cluster of nodes that can meet the input SLA. The possible crash of the Leader during a cluster configuration (or re-configuration) is detected via JGroups, and is dealt with by means of a simple recovery protocol which elects the MacroResourceManager with the currently smallest ID as the new Leader.

Note that the nodes in a cluster will host identical instances of the application, as homogeneous deployment is being carried out; hence, each node in that cluster can be used for honouring the application SLA. As the cluster is started up, the actual application can be deployed and run. Clients can issue RMIs to any node in the cluster, transparently.

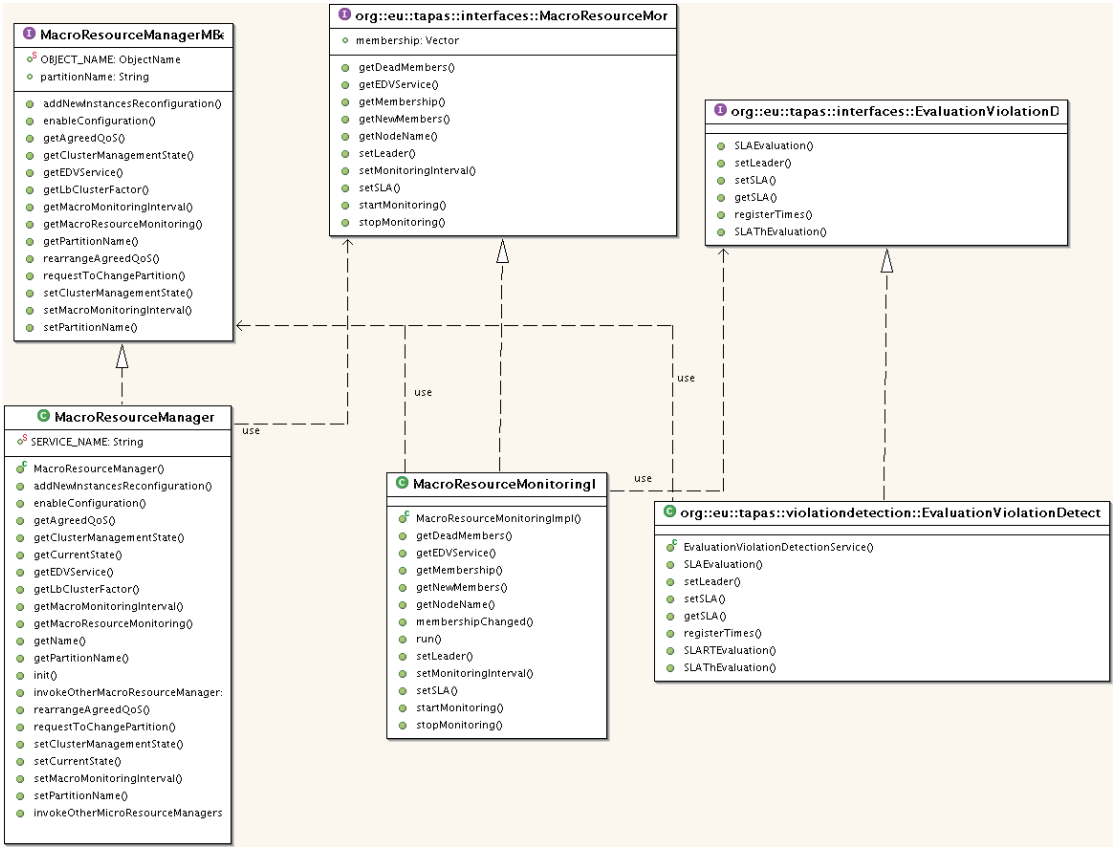

Figure 12: Macro Resource Manager and Macro Monitor interaction

If a failure occurs, (e.g., the crash of a JBoss node in the cluster), the standard failover mechanism in JBoss redirects the client RMIs, addressed to the crashed node, to another active node in the cluster. In the standard JBoss clustering service, this node will be selected according to one of the four load balancing policies introduced earlier, and specified at deployment time. As pointed out in Subsection 3.1.1, these policies select a target node with no knowledge of the run time computational load of

that node; hence, it is possible that the RMI redirection following a node failure in a cluster lead to overloading another node in that cluster. In principle, this process may continue until all nodes in that cluster are brought to an overloaded state, as a sort of domino effect. (Note that this process may defeat the adaptive HTTPLoadBalancer as well.)

In order to overcome this problem, in our implementation the CS aims to maintaining a fair distribution of the computational load among the clustered nodes. To this end, in case a node failure or an overload exception is raised by the MS within a cluster, our CS firstly attempts to reconfigure that cluster by integrating in it a spare node that replaces the faulty one; that spare node can be obtained possibly from another cluster (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 CS raises an exception to be dealt with at a higher level of abstraction (e.g., at the application level by adapting the application rather than the environment).

3.3 Load Balancing Service Implementation

As introduced in Section 3, we assume that client programs, typically browsers, interact with applications, hosted by clustered application servers, via HTTP requests. Hence, our Load Balancing Service implements load balancing of these requests among the clustered servers.

The current JBoss clustering mechanism provides support for load balancing of HTTP client requests. This mechanism makes use of the Apache web server with the mod_jk module, in order to balance the HTTP client requests at the web tier.

The mod_jk module is based on the notion of "workers"; i.e., clustered nodes toward which mod_jk forwards HTTP requests. This module must be statically configured, so as to specify the number of workers available in the cluster; its configuration cannot be changed dynamically (i.e., at run time).

This static configuration requirement represents a severe shortcoming of mod_jk, that prevents its usage for our purposes. In fact, we require dynamic cluster configuration and reconfiguration in order to, for example, recover from possible node failures, or replace overloaded nodes, at run time.

Moreover, if one of the Tomcat servers of the Web tier crashes, the mod_jk will not fail over to the remaining active nodes; thus, the active client sessions with the crashed node will be lost. (JBoss will be looking after the fail over of the HTTP client sessions.)

Owing to the above limitations of the JBoss load balancing, we have implemented a new load balancing service following the architecture described in Section 2. Figure 13 illustrates the UML diagram of our implementation.

The implementation consists of four principal Services, termed Load balancing Scheduler, HTTP Load Balancer, Request Interceptor, and Sticky Session Manager. These services, deployed in each node of the cluster (for survivability purposes), are described below.

The **Load Balancing Scheduler** (termed “LBScheduler” in the earlier Figure 5) is responsible for choosing the clustered target node to which a client request is to be forwarded. As discussed earlier, in order to choose the target node, this Scheduler

uses different load balancing policies, that can be plugged into the architecture dynamically, and are implemented by their relative policy managers, as shown in the UML diagram of Figure 9. We have currently included in our implementation (i) the *Adaptive Scheduler Manager*, which implements the WorkLoad policy introduced earlier, (ii) the *Random Scheduler Manager*, (iii) the *First Available Node Scheduler Manager*, and (iv) the *RoundRobin Scheduler Manager*.

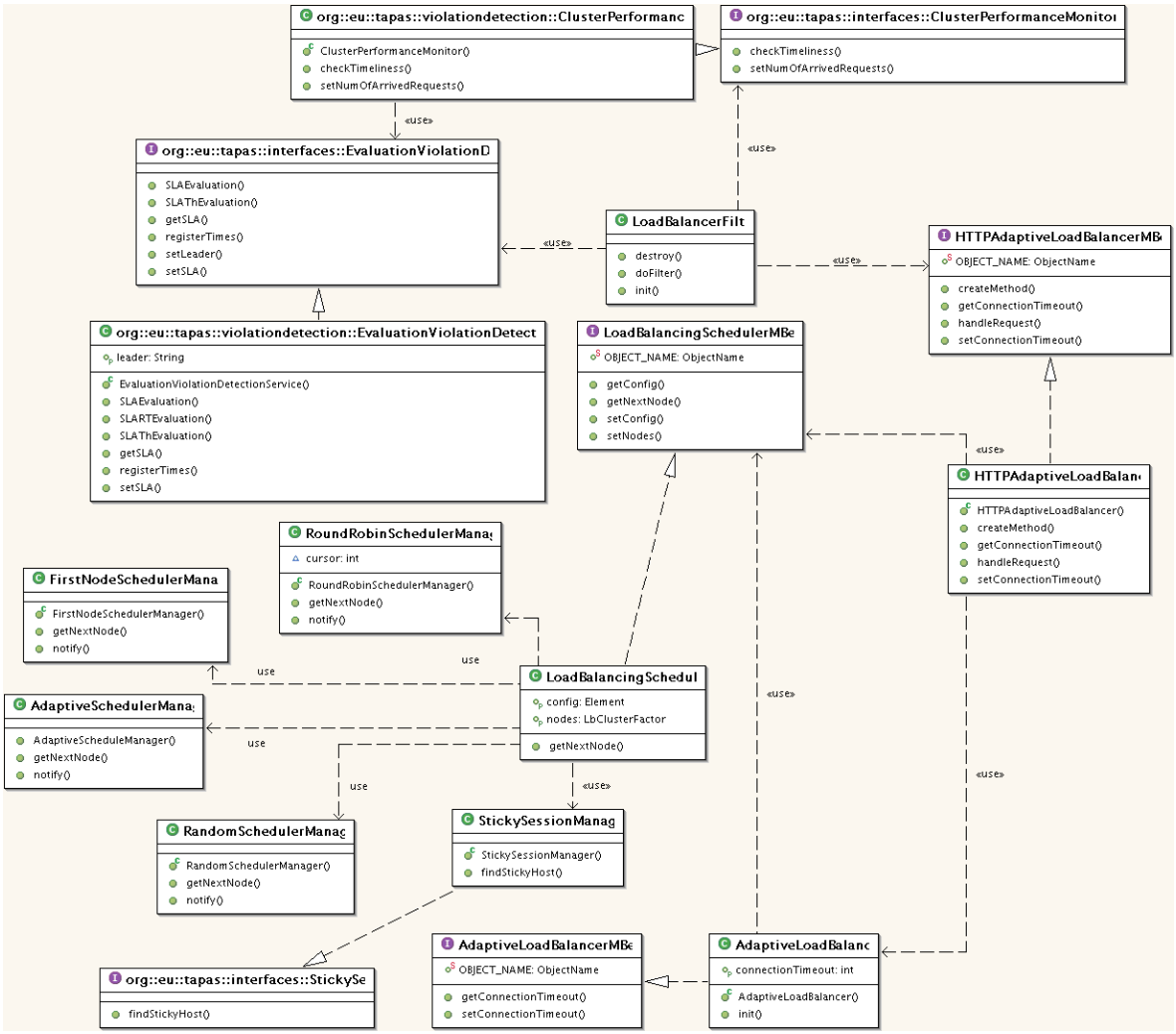

Figure 13:TAPAS load balancing implementation

The **HTTP Load Balancer** (i.e., the "HTTP Request Manager" in the earlier Figure 5) is used to balance the HTTP requests load, only. It uses the Load Balancing Scheduler Service to obtain the address of the target node that is to serve those requests, and implements the request manipulation protocol (i.e., the construction and forwarding of a new HTTP request) introduced earlier. To this end, the HTTP Load Balancer Service makes use of the AdaptiveLoadBalancer MBean. This MBean actually manages each HTTP request by directly invoking the Load Balancer Scheduler Service and building the new HTTP requests. (In practice, the AdaptiveLoadBalancer is a reverse proxy deployed in each JBoss node).

The client requests are intercepted by the **Request Interceptor** of our architecture, described in Section 2. This component is currently implemented by

using the Servlet Filter technology [12.a], provided by the JBoss embedded Tomcat web container. Figure 13 illustrates the LoadBalancerFilter class, and its principal methods.

The Filter dynamically intercepts each HTTP request, and either (i) applies the load balancing process to the request, or (ii) serves the request by passing it to the local web container. Specifically, when the filter intercepts a request, it first checks whether that request contains a special header we have created, termed "LoadBalanced" header. If that request does not contain this header, then it is being intercepted for the first time by the Filter. Hence, load balancing must be applied to that request (i.e., a target node that serve that request must be obtained from the Load Balancer Scheduler Service, and the request must be forwarded to that node). In contrast, if the request contains the "LoadBalanced" header, that request has been intercepted from a LoadBalancerFilter which has already applied load balancing to that request; hence, it is to be served. Thus, the request is passed to the local web container for processing.

The above mechanism implements the per-request load balancing introduced in Section 2 of this Report. As already discussed in that Section, this approach to load balancing can be expensive, in terms of performance, as it requires that mutual consistency among the clustered application servers be maintained (i.e., to this end, the JBoss HTTP session replication must be activated).

In order to assess a less expensive load balancing implementation, we have implemented a per-session load balancing which does not require the JBoss HTTP session replication. This form of load balancing can be selected at cluster configuration time, and is implemented by the **Sticky Session Manager**, depicted in Figure 13.

In essence, as the LoadBalancerScheduler selects a node to serve the first of a series of client requests (i.e., a client session), the Sticky Session Manager uses that node to serve all the following requests from that client, until the client session terminates. To this end, a unique "StickyHost" cookie, which identifies the IP address of the selected host, is generated at the time a client session starts, and maintained by the Sticky Session Manager, until that session ends. This cookie is used to identify the HTTP requests from the same client, and to associate always the same node to these requests.

In order to configure the JBoss application sever and the load balancing mechanism we have developed, we have created a so-called loadbalancing.properties file, deployed in the "conf" directory of our JBoss TAPASCluster configuration. This file contains an attribute termed "enable_loadbalancing" that is used to indicate whether or not our load balancing mechanism is to be enabled. In addition, a further configuration file has been created and deployed in the "deploy" directory of our JBoss TAPASCluster. The file, termed "tapasLoadBalancing-service.xml", is used to specify both (i) which Load Balancer Scheduler Manager is to be applied and (ii) whether or not the sticky session load balancing is to be used.

Finally, the filter declaration is not part of the JBoss TAPASCluster configuration. Rather, it is included into specific configuration files of the application deployed in the cluster. Specifically, the web.xml file of the distributed applications must contain the xml tags indicated in Figure 14.

```
<filter>
<filter-name>Load Balancer Filter</filter-name>
<filter-class>org.eu.tapas.loadbalancing.LoadBalancerFilter</filter-class>
</filter>

  <filter-mapping>
  <filter-name>Load Balancer Filter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

Figure 14: Servlet Filter declaration

4 Experimental Evaluation

In this Section we discuss the experimental results we have obtained from a set of preliminary tests of our middleware we have carried out. As already mentioned, further testing is currently in progress; the results of these additional tests are expected to be available by 31 March 2005, end date of the TAPAS project.

For our initial tests we have used a cluster configuration consisting of three application servers running on three Linux machines interconnected by a 100Mb Ethernet LAN. Each machine was 2.4 GHz Pentium 4, with 512MB of RAM. The three machines were dedicated to our experiments; the Ethernet LAN is part of our teaching laboratory, instead, and was used by our students while our experiments were in progress.

The Client was the JMeter program running on a G4 processor under Mac OS X, on the same LAN as the servers. All the results discussed below have been generated by JMeter. The application server cluster was hosting a digital bookshop application that provided its clients with conventional operations, such as “choose_book”, “add_book_to_cart”, “remove_book_from_cart”, “buy_book”, “confirm_order”. These operations accessed an application database; each client was performing 10 sessions, each of which consisted of 10 operations accessing that database. As the principal scope of this experimental evaluation was to assess the performance of our middleware services, only, the above mentioned database was replicated and instantiated locally to each application server, in order to avoid that it become a bottleneck; finally, issues of database consistency were ignored, for the scope of this evaluation.

Our first concern has been to assess whether our micro- and macro-resource levels were adding unnecessary overheads to the cluster response time (measured in ms) and throughput (measured in requests per second (rps)), in the absence of failures. To this end, we have instantiated our micro- and macro-resource levels in the cluster above, homogeneously hosting the digital bookshop application. The test consisted of 15 clients accessing this application, and performing the 10 sessions of 10 operations, as mentioned above (needless to say, this test was run numerous times; the results below are the average values we have obtained from the various runs). In view of the scope of this test (i.e., assessing the possible overheads caused by the micro and macro resource levels), load balancing was disabled.

The same test was repeated on a standard JBoss cluster (including no TAPAS middleware), with both the mod_jk load balancer and the server state replication disabled. The results of these tests are compared and contrasted in the following Figures 15 and 16.

These figures show that the overheads caused by the TAPAS middleware, compared to the JBoss standard middleware, are negligible, both in terms of response time and throughput. Specifically, Figure 15 shows that the TAPAS middleware introduces a 3% delay, approximately, in the response time; Figure 16 shows that the TAPAS cluster throughput is less the 1% inferior to that of a standard JBoss cluster.

The next test was intended to show the effectiveness of the TAPAS clustering mechanism in order to prevent SLA violations. Specifically, we have considered the following case. We have assumed that a possible ASP policy dictates that application deployment be carried out using the minimal set of resources which are required to run the application; hence, for example, an application is deployed on a single application server, if possible. A single application server may reach the response time and throughput breaching points, thus violating the hosting SLA, if it is not instrumented to reconfigure dynamically, prior to the SLA violation.

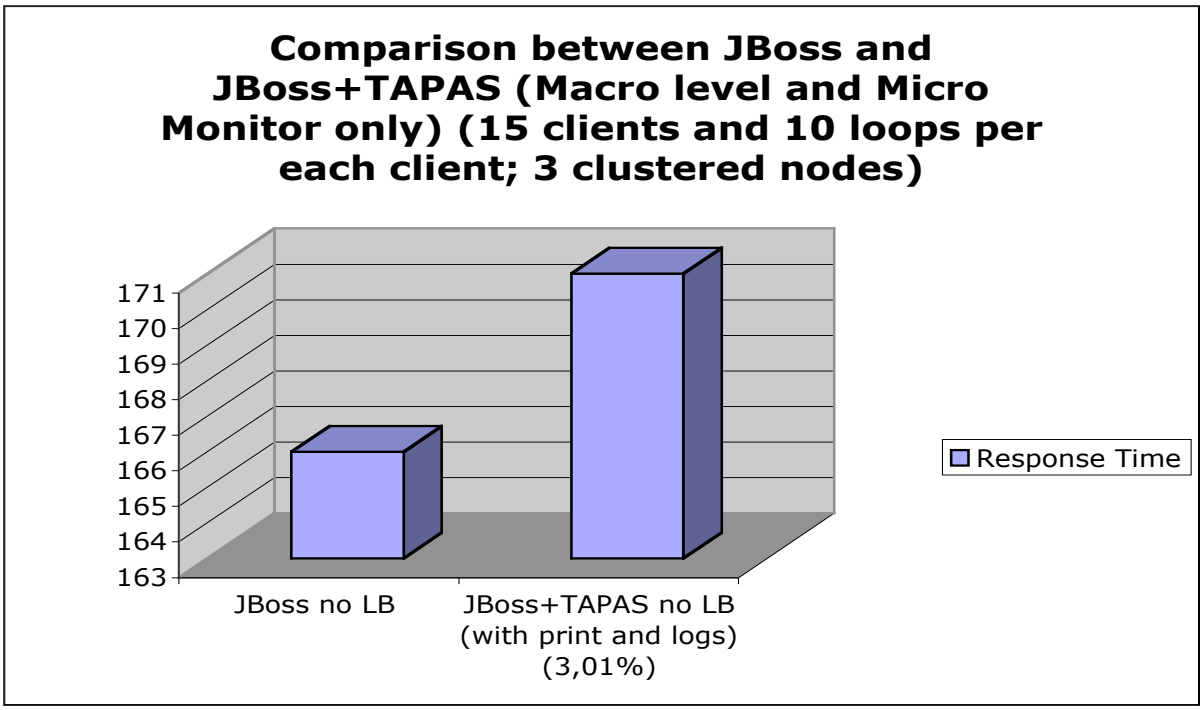

Figure 15: TAPAS response time overhead

Thus, we have compared and contrasted the deployment of our test application on a standard JBoss application server, equipped with mod-jk, with that of the same application on a TAPAS extended JBoss server, with per-session load balancing enabled, and two spare application servers for use for dynamic reconfiguration purposes. In this test, 50 clients were concurrently accessing the hosted application. The results of this test are depicted in Figures 17 and 18.

Figure 17 shows that the TAPAS middleware, with dynamic reconfiguration and load balancing, allows the application server to maintain the average response time below 850 ms. The standard JBoss, instead, provides an 18% slower response time than the TAPAS extended JBoss, approximately, as it is unable to apply reconfiguration, when necessary.

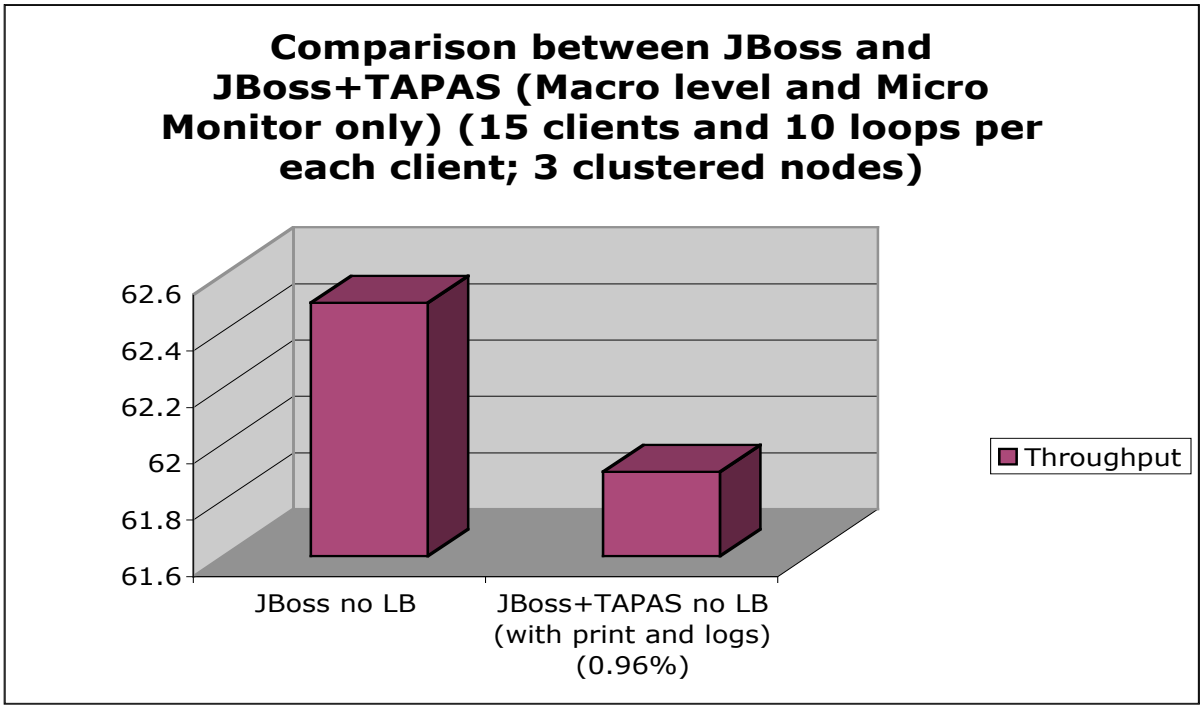

Figure 16: TAPAS throughput overhead

Similarly, Figure 18 shows that the standard JBoss throughput is approximately 14% lower than that generated by the JBoss application server extended with the TAPAS middleware (owing to the same motivation as above).

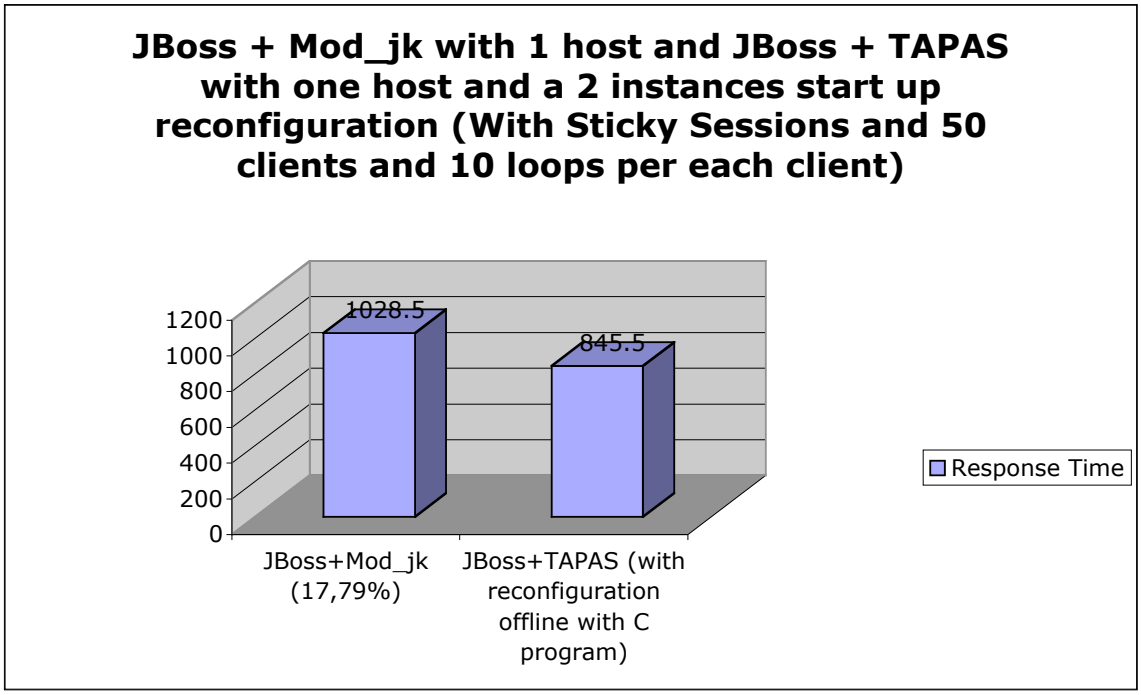

Figure 17: JBoss vs. TAPAS JBoss: response time

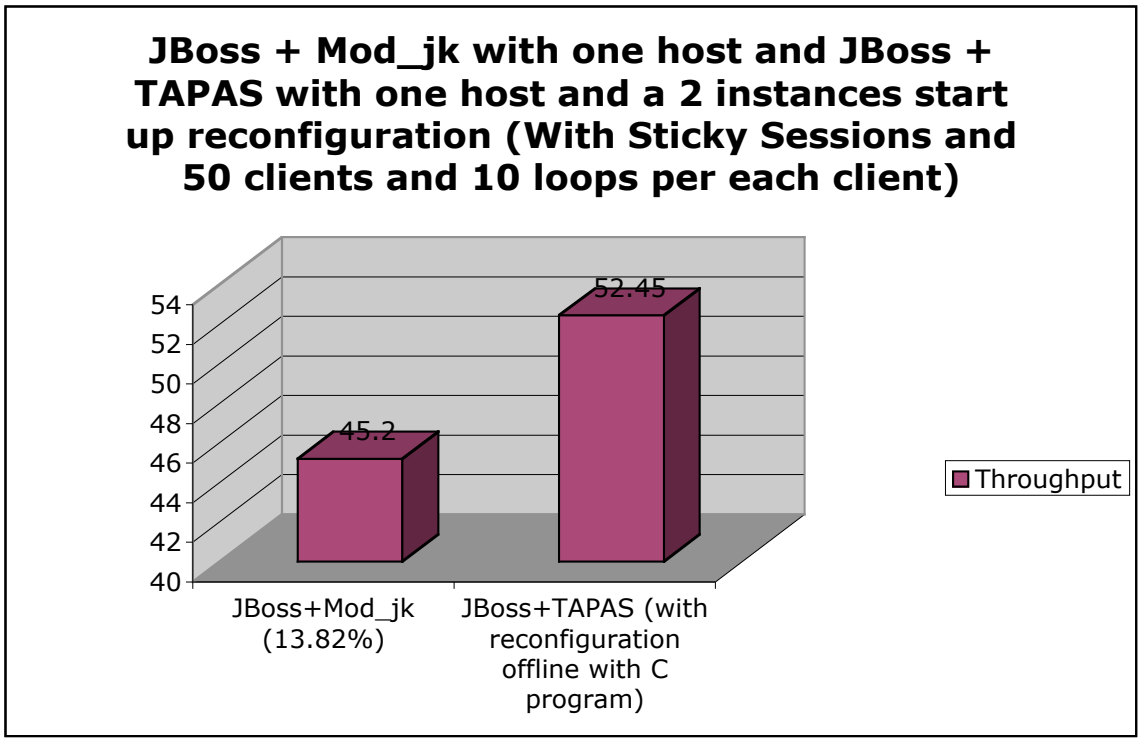

Figure 18: JBoss vs. TAPAS JBoss: throughput

We have carried out additional experiments using a different, more powerful, cluster of machines. This cluster consisted of 5 application servers running on 5 Linux machines interconnected by a 1Gb Ethernet LAN. One machine was dedicated to load balancing; the remaining 4 machines were used to serve client requests. Each machine in this cluster is a 2.66Ghz dual Intel Xeon processor, with 2GB of RAM. The five machines as well as the Ethernet LAN were dedicated to our experiments. As before, the Client program was the JMeter program running on a G4 processor under Mac OS X in the same LAN as the servers.

The scope of the first experiment was to assess the overhead, in terms of response time, caused by the overall TAPAS middleware, compared to that of the JBoss standard middleware. This experiment consisted of 50 clients accessing the same bookshop application introduced earlier, and performing 11 operations among those mentioned, each of which was repeated 10 times. Specifically, the experiment consisted of the following two tests.

The first test was performed on a standard JBoss cluster (including no TAPAS middleware), with Apache+mod_jk as load balancer; the second test was performed using our TAPAS middleware with the previously described load balancing service. Both tests measured the user perceived response time and the application throughput. The results of these tests are compared and contrasted in the following Figures 19 and 20.

Figure 19 shows that the TAPAS middleware causes a negligible response time overhead, when compared to the JBoss standard middleware. Specifically, Figure 19 shows that the TAPAS middleware introduces a delay that ranges from 2% to 8%, approximately, in case of one, two, or three nodes serving the client requests. In

contrast, in case four nodes are being used, the TAPAS middleware exhibits better response time (10% approximately) than the standard JBoss.

Figure 20 shows the throughput the TAPAS middleware and the standard JBoss can provide their clients with. This Figure shows that in case of two, three, and four nodes serving the client requests, the TAPAS middleware provides clients with higher throughput than the standard JBoss. Specifically, the throughput generated by the standard JBoss is up to 18% (approximately) lower than that produced by the JBoss application server extended with the TAPAS middleware. Instead, in case of one node, which serves the client requests, the TAPAS middleware throughput is lower than the one generated by the standard JBoss. However, in this case the difference is negligible (approximately 2%).

Note that, the aforementioned tests have been carried out using, in both the JBoss configurations, a load balancer that balances client sessions (i.e., a load balancer with sticky sessions). Hence, in those tests the JBoss provided HTTP session replication was disabled.

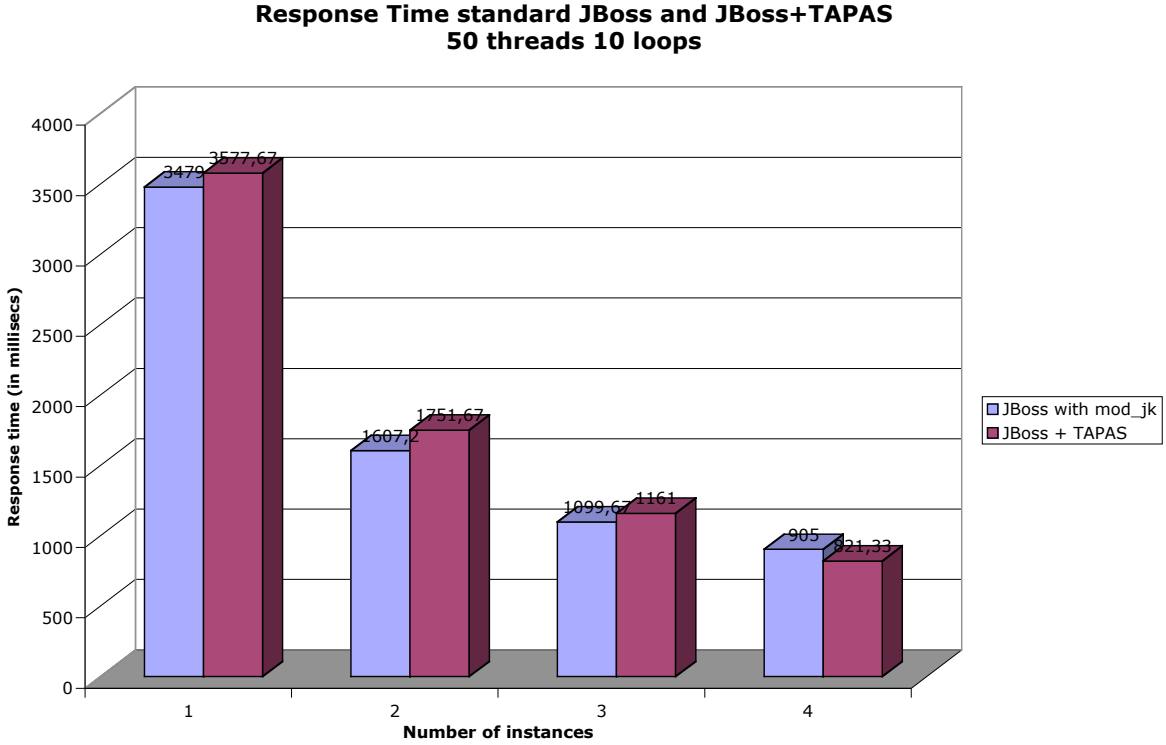

Figure 19: TAPAS vs JBoss+mod_jk Response time

The previous experiments show that the TAPAS middleware does not introduce a significant overhead compared to the JBoss standard configuration. The next experiments show that TAPAS middleware can honour the response time specified in the SLA.

We have performed two tests. In both tests we have assumed that the SLA specifies that the HTTP request user perceived response time is to be below 1s. In the first test we have imposed a load of 10, 20 and 30 clients on a TAPAS cluster consisting of one node. As the load augments, the TAPAS middleware reconfigures the cluster adding new instances (up to 4), in order to honour the SLA. We have

imposed the same load on a JBoss cluster composed by a single node. In contrast to the TAPAS middleware, the standard JBoss does not reconfigure the cluster, and maintains only the initial instance during the experiment.

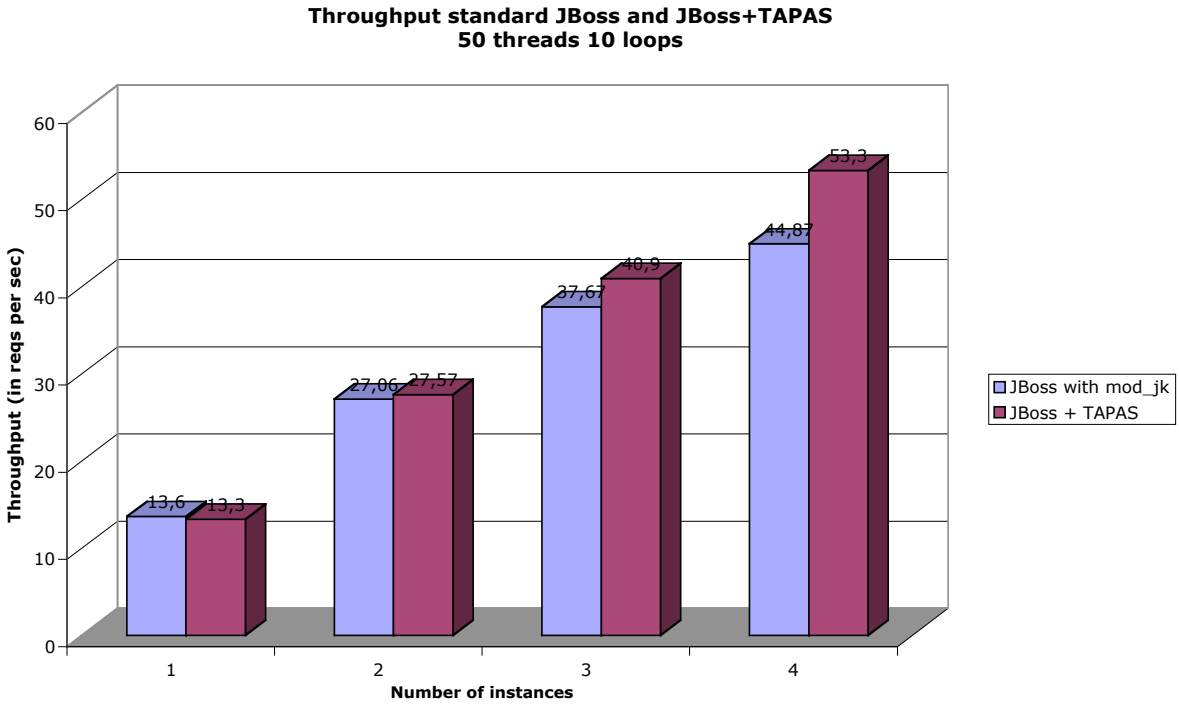

Figure 20: TAPAS vs JBoss+mod_jk Throughput

As we can see from the Figure 21, if the load is low (10 clients) the response time is similar in both the configurations. A higher load (20 and 30 clients) shows a notable difference between the two configurations. With 20 and 30 clients, the JBoss response time is, respectively, 124% and 362% higher than that obtained by the TAPAS middleware. Note that Figure 21 shows that the TAPAS middleware response time with 10 or 20 clients is approximately constant. In contrast, as the number of clients grows to 30, the reponse time reduces to 444 ms, on average. This is the effect of the reconfiguration that occurred during the test, as new instances were automatically added to the initial configuration, and the requests load was balanced among a number of nodes larger than that of the initial configuration.

The second test we have performed is similar to the one above, with the exception that the initial configuration is a cluster of two nodes (both testing standard JBoss and TAPAS middleware). We have imposed a load of 10, 20, 30, 40, and 50 clients. As in the previous test, the JBoss standard cluster maintains the initial instances only; in contrast, the TAPAS middleware adds instances dynamically (up to 4) in order to meet the response time requirement. As expected, the Figure 22 shows that for load of 10 and 20 clients we obtain the same response time in both JBoss and TAPAS (this is because both configuration have the same number of instances), while when the load augments (30, 40, 50 client) the response time of the TAPAS middleware is respectively 39%, 48%, 50% lower than that of the standard JBoss.

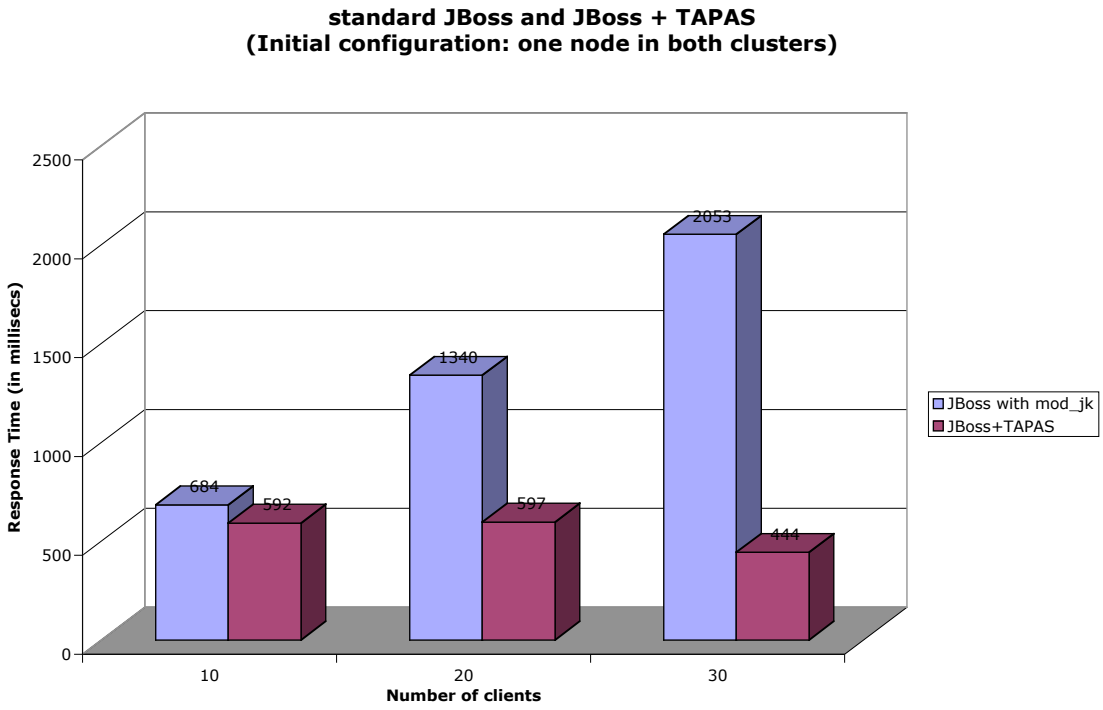

Figure 21: TAPAS vs JBoss+mod_jk (single node cluster)

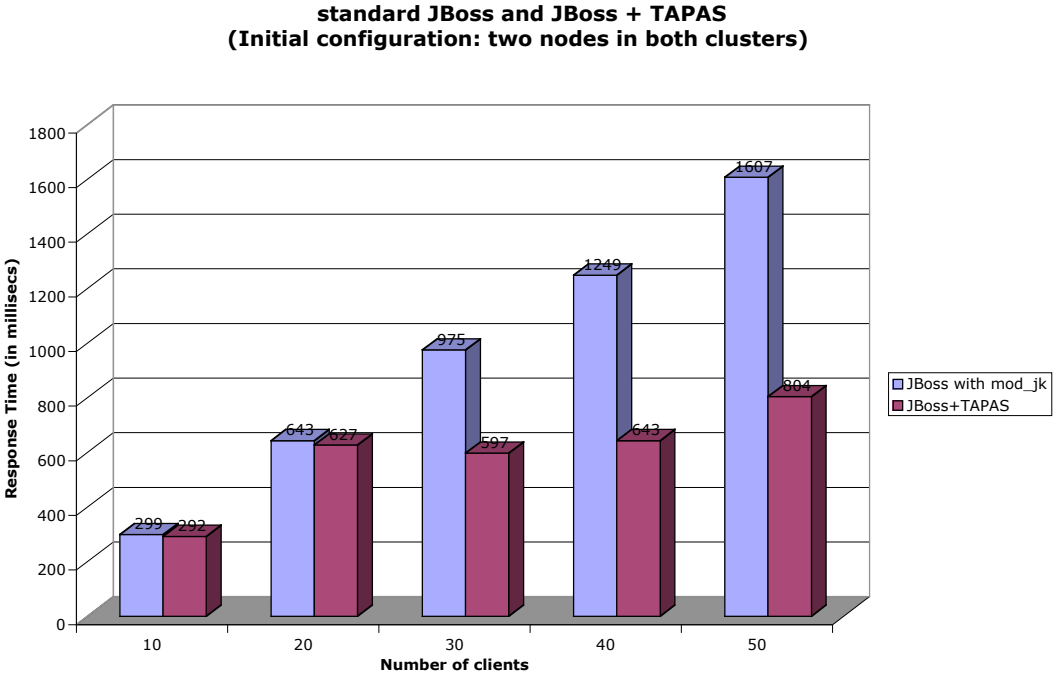

Figure 22: TAPAS vs JBoss+mod_jk (two nodes cluster)

5 Related Work

The research activity on QoS enforcement, monitoring, and resource clustering, in the context of both Web services and application server technologies, has been very

active in recent years. This activity has produced a number of relevant proposals which have influenced notably our approach to the design of a QoS-aware application server. In this Section, we review some of these proposals, and compare and contrast them with our own, described in this Report.

In [30], the authors discuss the design of a framework that enables the development of composite Web services. Their proposal is based on the use of forward error recovery techniques, at the compositional level, in the specification of the behaviour of composite Web services in the presence of failures. These services can be structured as coordinated atomic actions which have a well-defined behaviour, both in the presence and absence of failures.

In [18], the principal issues involved in the design of high-volume, highly dependable Web services are discussed. In particular, this paper describes a number of both hardware (e.g., Cisco's DistributedDirector), and network-based (e.g., NAT, HTTP redirect, DNS round-robin) solutions that can be deployed in order to meet Web service scalability requirements, and summarizes a variety of software-implemented techniques (e.g., transactions on objects and process groups) that can be used in order to meet Web service fault-tolerance requirements.

The main difference between the above works and the application server design and implementation discussed in this Report is that we have developed specific services, at the middleware level, that can support a generic, dependable application hosting in the context of component-based technologies, rather than just Web services.

The paper [8] is strictly related to this class of technologies; this paper describes a new model of QoS-aware components and QoS-aware containers (i.e. run-time environments whereby QoS-aware components execute). In particular, this paper proposes to enhance those containers with facilities for QoS configuration and QoS negotiation.

Conan et al. [5] describe generic mechanisms that enable the use of containers together with aspect-oriented software development (AOSD) techniques, in order to plug in different non-functional properties.

The above two approaches differ from our approach as they provide new types of application components, and a new run-time environment for those components, in order to meet non-functional application requirements. In contrast, we have designed and implemented an extension of an open source application server that augments that server with QoS capabilities, transparently to the application components.

Issues of resource clustering have been widely investigated in the literature, since a number of years. Recent relevant work includes the following. In [31] techniques are described for the provision of CPU and network resources in shared hosting platforms (i.e. platforms constructed out of clusters of servers), running potentially antagonistic third-party applications. In particular, the architecture proposed in this paper provides applications with performance guarantees by overbooking clustered resources, and by combining this technique with commonly used resource allocation mechanisms.

This work is similar to that discussed in [1], which describes a comprehensive framework for resource management in Web servers, with the aim of delivering predictable QoS and differentiated services. However, in this paper application

overload is detected when resource usage exceeds some predetermined thresholds; in contrast, in [31], the application overload is detected by observing the tail of recent resource usage distributions.

In [26] the design and implementation of an integrated resource management framework for cluster-based network services is investigated. Specifically, an adaptive multi-queue scheduling scheme is employed inside each node of a clustered environment, in order both to achieve efficient resource utilization under quality constraints, and to provide service differentiation.

IBM Lotus [10] provides clustering features that can be applied to wide area network based clusters for some such Lotus services as Lotus Instant Messaging and Web Conferencing.

Finally, current application server technologies, both open-source (e.g., JBoss [13], JOnAS [16]) and proprietary (e.g., WebLogic [3], WebSphere [11]), implement clustering services that provide the hosted applications with cluster-wide load balancing, fail-over, state transfer, and cluster membership management functionalities. These services exhibit some shortcomings, (e.g., lack of dynamic load balancing) discussed earlier, and do not address issues of SLAs enforcement and monitoring.

In contrast, these issues are discussed in [20] and [7]. These papers propose specific SLAs for Web services, and focus on the design and implementation of an SLA compliance monitor, possibly owned by Trusted Third Parties (TTPs). In addition, [33] presents an architecture for the coordinated enforcement of resource sharing agreements (i.e., SLAs) among applications using clustered resources. The approach to the design of this architecture shares a number of similarities with our approach (e.g., the resource sharing among the applications is governed by the SLAs these applications have with their hosting environment). However, prototype implementations of this architecture have been developed both at the HTTP level, and at the transport level, rather than at the middleware level, as in our approach.

In summary, from the above discussion it emerges that clustering provides one with an indeed valuable approach to the design and development of adequate support for highly available distributed applications, as these applications can be replicated across multiple machines, in a controlled manner. However, to the best of our knowledge, none of the clustering mechanisms cited above is based on augmenting existing open-source component technology with middleware services for cluster-wide SLA enforcement, configuration, run-time monitoring of the delivered QoS, and adaptation, as in our approach.

6 Concluding Remarks

In this Report we have described the design and implementation of a collection of middleware services we have developed in order to construct what we have termed a QoS-aware application server. These services have been implemented as an extension of the open-source JBoss application server. As JBoss offers a clustering service, our work has included the design and implementation of a QoS-aware clustering service.

In addition, in this Report we have discussed the results of a preliminary evaluation of our implementation. These results show the effectiveness of our

approach. An additional evaluation of our work is currently in progress, and we expect to report on it at the time of the Tapas project final review (31 march 2005).

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, and described the implementation of an adaptive load balancing policy we have carried out.

In addition, in this Report we have pointed out the following three principal 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 consists of 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-cluster 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 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).

For the purposes of the work described in this Report, we have used clustered machines interconnected by a LAN. However, in principle, a 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 investigating issues of QoS-aware geographical clustering across the Internet, using a specific VPN technology, named VDE [6], which has been developed in our Department. This technology provides its users (i.e., the application servers, in our case) with the abstraction of an overlay Ethernet LAN, constructed on top of the Internet. We are planning to use this technology in order to evaluate the performance of our clustering mechanism when deployed in a geographical context. In addition, we are planning to investigate issues of homogeneous and heterogeneous application deployment, in the context of a geographically distributed cluster of application servers.

References

[1] M. Aron, P. Druschel,and W. Zwaenepoel, ``Cluster Reserve: A Mechanism for Resource Management in Cluster-based Network'', in Proc. ACM SIGMETRICS Conference, Santa Clara, CA, June 2000.

[2] ASP Industry Consortium White Papers, "SLA for Application Service Provisioning", http://www.allaboutasp.org.

[3] BEA, ``BEA WebLogic Server 8.1 Overview: The Foundation for Enterprise Application Infrastructure'', White Paper, August 2003.

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

[5] D. Conan, E. Putryez, N. Farcet and M. A. de Miguel, ``Integration of Non-Functional Properties in Containers'', in Proc. 6th International Workshop on Component-Oriented Programming, Budapest, Hungary, 2001.

[6] R. Davoli,``VDE: Virtual Distributed Ethernet'', Technical Report TR UBLCS-2004-12, Department of Computer Science, The University of Bologna, June 2004.

[7] M. Debusmann, A. Keller, ``SLA-driven Management of Sistributed Systems using the Common Information Model'', in Proceedings of the 8th International IFIP/IEEE Symposium on Integrated Management (IM 2003), Colorado Springs, CO, USA, March 2003.

[8] Miguel A. de Miguel, ``QoS-Aware Component Frameworks'', in Proceedings of the 10th International Workshop on Quality of Service - IWQoS2002, Florida, 2002.

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

[9.a] G. Ferrari, S. Shrivastava, P. Ezhilchelvan, "An Approach to Adaptive Performance Tuning of Application Servers", in Proc. 23rd Symp. on Reliable and Distributed Systems (SRDS 2004), Florianopolis (Brazil), 18 – 20 October, 2004.

[10] http://www-106.ibm.com/developerworks/lotus

[11] http://www-306.ibm.com/software/webserver/appserv

[12] http://java.sun.com/j2ee/ecperf/index.jsp

[12.a] http://java.sun.com/products/servlet/docs.html

[13] http://www.jboss.org

[14] http://www.jgroups.org/javagroupsnew/docs

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

[16] http://www.objectweb.org

[17] http://wwws.sun.com/software/products/

[18] David B. Ingham, Santosh K. Shrivastava and Fabio Panzieri, ``Constructing Dependable Web Services'', IEEE Internet Computing, Vol 4 n.1:25-33, January-February, 2000.

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

[20] A. Keller and H. Ludwig, ``The WSLA Framework: Specifying and Monitoring Service Level Agreements for Web Services'', Journal of Network and Systems Management, Special Issue on E-Business Management, Vol. 11, N. 1, Plenum Publishing Corporation, Presprint available as IBM Reseach Report RC22456, March 2003.

[21] S. Labourey and B. Burke, ``JBoss Clustering 2nd Edition'', 2002.

[22] S. Labourey “Load Balancing and Failover in the JBoss Application Server”, 2001-2004 IEEE Task Force on Cluster Computing, http://www.clustercomputing.org
[23] S.Labourey and B.Burke “ JBoss Clustering 2nd Edition”, 2002.
[24] C. Molina-Jimenez, S. Shrivastava, J. Crowcroft and P. Gevros, ``On the Monitoring of Contractual Service Level Agreements'', 1st IEEE International Workshop on Electronic Contracting (WEC), July 2004, San Diego.
[25] B. Shannon, Java 2 Plaftorm Enterprise Edition v. 1.4, Sun Microsystem, Final Release 24 November 2003.
[26] K. Shen, H. Tang, T. Yang and L. Chu, ``Integrated Resource Management for Cluster-based Internet Services'', in Proc. 5th Symposium on Operating Systems and Design and Implementation, USENIX Association, Boston Massachussets, USA, December 9-11 2002.
[27] S.K. Shrivastava, “An Overview of the TAPAS Architecture”, TAPAS Deliverable Report D5-extra, July 2003.
[28] J. Skene, D. Lamanna and W. Emmerich ``Precise Service Level Agreements'' in Proc. 26th International Conference on Software Engineering (ICSE'04), Edinburgh, Scotland (UK), 25 May 2004 (to appear).
[29] Sun Microsystem, ``Java Management eXtension: Instrumentation adn Agent Specification v.1.1'', 2002. Avalable at http://java.sun.com/jmx
[30] F. Tartanoglu, V. Issarny, A. Romanovsky, and N. Levy, ``Coordinated Forward Error Recovery for Web Services'', in Proceeding of the 22nd Symposium on Reliable Distributed Systems (SRDS'2003), October 2003, Florence, Italy.
[31] B. Urgaonkar, P. Shenoy and T. Roscoe, ``Resource Overbooking and Application Profiling in Shared Hosting Platforms'', in Proc. 5th Symposium on Operating Systems and Design and Implementation, USENIX Association, Boston Massachussets, USA, December 9-11 2002.
[32] N. Wang, D. C. Schmidt and C. O'Ryan, ``An Overview of the CORBA Component Model'', in Computer-Based Sofware Engineering (G. Heineman and B. Councill, Eds.), Reading, Massachusetts: Addison-Wesley, 2000.
[33] T. Zhao and V. Karamcheti, ``Enforcing Resource Sharing Agreements among Distributed Server Clusters'', in Proceedings of the 16th International Parallel and Distributed Processing Symposium (IPDPS), April 2002.
[34] Davide Rossi, Elisa Turrini , "Testing J2EE clustering performance and what we found there", Proc. 1st IEEE International Workshop on Quality of Service in Application Servers (QoSAS 2004), in conjunction with 23rd Symposium on Reliable Distributed Systems (SRDS 2004), Jurerê Beach Village, Santa Catarina Island, Brazil, 17 October 2004.

Appendix

This Appendix contains the following four papers:

1 Giorgia Lodi, Fabio Panzieri, "QoS-aware Clustering of Application Servers", Proc. 1st IEEE International Workshop on Quality of Service in Application Servers (QoSAS 2004), in conjunction with 23rd Symposium on Reliable Distributed Systems (SRDS 2004), Jurerê Beach Village, Santa Catarina Island, Brazil, 17 October 2004.

2 Davide Rossi, Elisa Turrini , "Testing J2EE clustering performance and what we found there", Proc. 1st IEEE International Workshop on Quality of Service in Application Servers (QoSAS 2004), in conjunction with 23rd Symposium on Reliable Distributed Systems (SRDS 2004), Jurerê Beach Village, Santa Catarina Island, Brazil, 17 October 2004.

3 Giovanna Ferrari, Santosh Shrivastava, Paul Ezhilchelvan, "An Approach to Adaptive Performance Tuning of Application Servers", Proc. 1st IEEE International Workshop on Quality of Service in Application Servers (QoSAS 2004), in conjunction with 23rd Symposium on Reliable Distributed Systems (SRDS 2004), Jurerê Beach Village, Santa Catarina Island, Brazil, 17 October 2004.

4 Paul Ezhilchelvan, Giovanna Ferrari, Mark Little, "Realistic and Tractable Modeling of Multi-tiered E-business Service Provisioning", Technical Report, The University of Newcastle upon Tyne, March 2005.