# REQUIREMENT ANALYSIS FOR CLUSTERING APPLICATION SERVERS

Mr. Nagesh V
Lecturer, Dept. of Computer Science
Atria Institute of Technology
AIKBS , Campus, AGS Colony
Bangalore , India
Email : nageshunkt@gmail.com

***Abstract***— In this paper, we discuss the design, implementation, and experimental evaluation of a middleware architecture for enabling Service Level Agreement (SLA)-driven clustering of QoS-aware application servers. Our middleware architecture supports application server technologies with dynamic resource management: Application servers can dynamically change the amount of clustered resources assigned to hosted applications on-demand so as to meet application-level Quality of Service (QoS) requirements. These requirements can include timeliness, availability, and high throughput and are specified in SLAs. A prototype of our architecture has been implemented using the open-source J2EE application server JBoss. The evaluation of this prototype shows that our approach makes possible JBoss' resource usage optimization and allows JBoss to effectively meet the QoS requirements of the applications it hosts, i.e., to honor the SLAs of those applications.

***Index Terms***— Service Level Agreement, Quality of Service, QoS-aware application server, QoS-aware cluster, dynamic cluster configuration, monitoring, load balancing

## I. INTRODUCTION

Distributed enterprise applications (e.g., stock trading, business-to-business applications) can

be developed to be run with application server technologies such as Java 2 Enterprise Edition (J2EE) servers, CORBA Component Model (CCM) servers, or .NET. These technologies can provide the applications they host with an execution environment that shields those applications from the possible heterogeneity of the supporting computing and communication infrastructure; in addition, this environment allows hosted applications to openly access enterprise information systems, such as legacy databases.

These applications may exhibit strict Quality of Service (QoS) requirements, such as timeliness, scalability, and high availability that can be specified in so-called Service Level Agreements (SLAs). SLAs are legally binding contracts that state the QoS guarantees an execution environment has to supply its hosted applications.

Current application server technology offers clustering and load balancing support that allows the application designer to handle scalability and high availability application requirements at the application level; however, this technology is not fully tailored to honor possible SLAs.. In order to overcome this limitation, we have developed a middleware architecture that can be integrated in an application server to allow it to honor the SLAs of the applications it hosts—in other words, to make it QoS-aware. The designed architecture supports dynamic clustering of QoS-aware Application Servers (QaASs) and load balancing. In current J2EE servers, the clustering support is provided in the form of a service. In general, that service requires the initial cluster configuration to consist of a fixed set of application server instances. In the case of peak load conditions or failures, this set of instances can be changed at runtime by a human operator reconfiguring the cluster as necessary (e.g., by introducing new server instances or by replacing failed instances). In addition, current clustering support does not include mechanisms to guarantee that application-level QoS requirements are met. These limitations can impede the efficient use of application server technologies in a utility computing context. In fact, current clustering design requires overprovision policies to be used in order to cope with variable and unpredictable load and prevent QoS requirements violations.

Our middleware architecture is principally responsible for the dynamic configuration, runtime monitoring, and load balancing of a QoS-aware cluster. It operates transparently to the hosted applications (hence, no modifications to these applications are required) and consists of the following three main services: Configuration Service, Monitoring Service, and Load Balancing Service.

### 1.1 MIDDLEWARE PLATFORM

A middleware platform is generally used as an architectural component for supporting the development and the execution of distributed applications. Its main role is to create a level of abstraction so as (i) to present a unified programming model to application developers and (ii) to mask out problems of system and network heterogeneity. Middleware can be composed by multiple layers. There can be identified four principal levels

• **Host Infrastructure** Middleware it encapsulates and enhances native operating system communication and concurrency mechanisms to create portable and reusable network programming components;

• **Distribution** Middleware it defines higher-level distributed programming models whose reusable APIs and mechanisms automate the native operating system network programming capabilities encapsulated by the previous level

• **Common** Middleware Services the collection of the services of this level are responsible for augmenting the distribution middleware layer by defining higher-level domain-independent components that allow the application designers to concentrate on the application logic only;

• **Domain-specific** Middleware Services these services are tailored to the requirements of a specific application domain and embody knowledge of that domain.

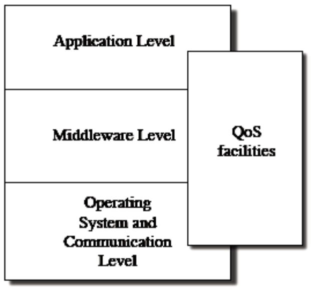

**Figure 1. Levels of QoS Integration**

Nowadays the middleware technology is largely adopted, in order to make easier the development of distributed applications; however, it is important that the middleware remains effective for such types of applications (e.g., enterprise applications) that can impose demands in terms of resource availability, adaptivity, reliability, scalability, and timeliness. In fact, these applications must operate under changeable environment conditions and they present stringent Quality of Service (QoS) requirements that are to be met in order to guarantee the correct behavior of the applications themselves.

Figure 1 depicts the levels of the software infrastructure in which a QoS management system should be provided. Thus, for example, at the operating system level, there should be mechanisms for reserving such resources as CPU, memory and threads; the communication level should provide applications with mechanisms for network monitoring and reservation; the middleware level should be constructed out of services for QoS negotiation, monitor an adaptation and finally QoS monitoring and adaptation can be applied at the application level as well, by allowing this level to monitor and adapt the QoS it may require.

## 2 SERVICE LEVEL AGREEMENTS

In current industrial practice, QoS requirements are specified in so-called SLAs.

Our SLA represents a collection of contractual clauses binding a QoS-aware cluster to the applications it hosts. We term this SLA a hosting SLA. This is an XML file that consists of two principal sections: Client Responsibilities and Server Responsibilities. These define the rights and obligations of the application clients and the application server, respectively. Both the Client and Server Responsibilities may specify different levels of QoS, each related to some (or all) operations of the hosted application. Hence, a client obligation could specify the maximum number of requests clients are allowed to send to the application, within a defined time interval.

The following SLA fragment shows the requestRate, which serves to capture this specific client obligation. The fragment is part of a larger hosting SLA example for a conventional bookshop application. It provides clients with operations such as “login,” “catalog,” “bookDetails,” “addToCart,” and so on.

```
<ContainerServiceUsage name=”HighPrority”
                    RequestRate=”100/s”>
<Operations>
<Operation path=”catalog.jsp” />
<Operation path=”AddToCart” />
<Operation path=”checkout.jsp” />
<Operation path=”CheckoutCtl” />
</Operations>
...
</ContainerServiceUsage>e allows a J2EE cluster to react to
```

Server obligations may include service availability guarantees. The fragment of the hosting SLA below shows possible availability guarantees for customers of a typical bookshop application.

```
<ServerResponsibilities
               serviceAvailability=”0.99”
                       efficiency=”0.95”
                 efficiencyValidity=”2”>
<OperationPerformance name=”HighPriority”
                maxResponseTime=”1.0s”>
<Operations>
<Operation path=”catalog.jsp” />
<Operation path=”AddToCart” />
<Operation path=”checkout.jsp” />
<Operation path=”CheckoutCtl” />
</Operations>
</OperationPerformance>
...
</ServerResponsibilities>
```

The serviceAvailability attribute specifies the probability with which the hosted application must be available over a predefined time period. In addition, each application operation specified as part of the SLA Server Responsibilities can be lassified according to a QoS attribute. In the example above, we opted for the response time attribute maxResponseTime, as it is used in most commercial SLAs (e.g., [1], [49], [33]) as an effective parameter for measuring service responsiveness. Finally, as pointed out in [9], the SLA may also specify the percentage of SLA violations that can be tolerated, within a predefined time interval, before the application service provider incurs a (e.g., economic) penalty.

## 3 THE MIDDLEWARE ARCHITECTURE

We have identified the following three main issues in the design of our architecture:

1. Guaranteeing that the QoS requirements specified in SLAs are met.
2. Optimizing the resource utilization in addressing item 1, above.
3. Maximizing the portability of the software architecture across a variety of specific J2EE implementations.

To address these issues, we conducted an in-depth assessment of the state-of-the-art in the design of architectures developed to meet the QoS requirements of distributed applications. This helped us to formulate a number of recommendations and principles that guided our design. Therefore, for example, these recommendations include the need for a resource monitoring service that assesses the resource state at runtime; the design of dynamic adaptation facilities was based on principles derived from the feedback control theory [35]. In addition, as we are dealing with a clustered environment characterized by highly variable and unpredictable load conditions, dynamic load balancing mechanisms may be necessary. These mechanisms allow us to balance client requests among clustered servers, based on the actual load of those servers, thus preventing server overloading.

In view of the above observations, we designed a middleware architecture incorporating three principal QoS-aware middleware services: a Configuration Service, a Monitoring Service, and a Load Balancing Service.

As already mentioned, this architecture is designed to be deployed in a cluster of application servers. The cluster consists of application server instances (termed nodes). Each node hosts a replica of our services; our architecture implements a primary-backup replication scheme [11] for fault-tolerance purposes.

The principal responsibilities of the three services mentioned above can be summarized as follows:

**The Configuration Service** is responsible for configuring the QoS-aware cluster so it can meet the customer application hosting SLA. The main activities performed by the Configuration Service include configuring the cluster at the time the hosting SLA is deployed in the QoS-aware cluster (at SLA deployment time) and possibly reconfiguring the cluster at runtime.

The cluster configuration process consists of building the initial cluster by forming a group of nodes from a minimal set of available nodes to ensure the service availability requirement of the hosting SLA is met.

The runtime reconfiguration process consists of dynamically resizing the cluster configuration, by adding or removing clustered nodes, as needed. Adding nodes can be necessary in order to handle a dynamically increasing load and in case a clustered node fails and needs to be replaced by an operational one (or possibly more than one); for this purpose, a pool of spare nodes is maintained.

Releasing nodes may be necessary to optimize the use of the resources. If the load on a hosted application significantly decreases, some of the nodes allocated to that application can be dynamically deallocated and included in the pool of spare nodes for further usage.

**The Monitoring Service** is in charge of monitoring the QoS-aware cluster at application runtime so as to detect possible 1) variations in the cluster membership, 2) variations in cluster performance, and 3) violations of the hosting SLA.

Thus, the Monitoring Service periodically checks the cluster membership configuration to detect whether clustered nodes should join or leave the cluster following failures or voluntary connections to (or disconnections from) the cluster. In addition, it monitors data such as cluster response time, client request rate, and cluster SLA violations to detect whether the cluster-delivered QoS deviates from what is required and specified in the hosting SLA. Specifically, this service makes use of a collection of parameters computed and updated at run time. These parameters allow he Monitoring Service to keep track of the dynamic behavior of the cluster in order to check whether or not the cluster is honoring the hosting SLA at runtime; they serve to maintain 1) the cluster's operational conditions trend, 2) the operational conditions trend of each clustered node, and 3) the cluster violation rate trend.

**The Load Balancing Service** is implemented at the middleware level and balances the load of HTTP client requests among the clustered nodes; it contributes to meeting the hosting SLA by preventing the occurrence of node overload and avoiding the use of resources that have become unavailable (e.g., failed) at runtime. The reason for implementing load balancing at the middleware level is twofold; namely, implementing load balancing at this level allows independence from any underlying operating system. In addition, the designed Load Balancing Service can easily detect specific application server conditions, such as server response time and cluster membership configuration. The Load Balancing Service we have developed can be thought of as a reverse proxy server that essentially intercepts client HTTP requests for an application and dispatches these requests to the nodes hosting that application. It includes support for both request-based and session-based load balancing. With request-based load balancing, each individual client request is dispatched to any clustered node for processing; in contrast, with sessionbased load balancing, client requests belonging to a specific client session are dispatched to the same clustered node.

The Load Balancing Service is responsible for

1. intercepting each HTTP client request,
2. selecting a target node that can serve that request by using specific load balancing policies,
3. deftly manipulating the client request to forward it to the selected target node,
4. receiving the reply from the selected target node, and, finally,
5. providing a reply to the client who has triggered the request.

The load balancing policy embodied in our Service (termed WorkLoad policy) is an adaptive policy, as we are interested in dynamically balancing the load among clustered nodes. This policy enables the Load Balancing Service to select a lightly loaded node among those in the cluster in order to serve client requests.

### 3.1QoS-Aware Middleware Services Interactions

Our QoS-aware middleware services cooperate with each other to ensure hosting SLA enforcement and monitoring. Fig. 2 shows how they interact.

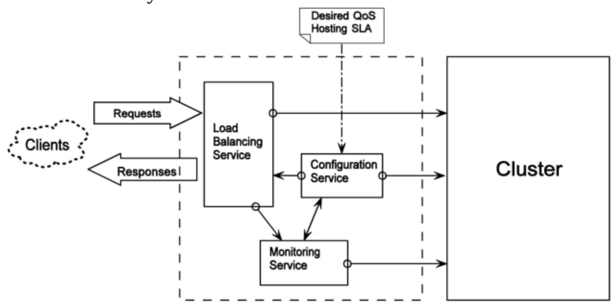

**Fig 2 QoS-Aware Middleware Services Interactions**

In Fig. 2, client requests are intercepted by the Load Balancing Service. For each request, the QoS delivered by the cluster is compared to the desired level of QoS specified in the hosting SLA in order to monitor adherence to this SLA. To this end, the Configuration Service makes the hosting SLA content available to the Monitoring Service. The Monitoring Service cooperates with the Load Balancing Service to obtain the QoS delivered by the cluster. Based on the retrieved QoS data, the Monitoring Service computes and updates the monitoring parameters (see Section 4), which serve to check whether the cluster operational conditions are close to violating the hosting SLA. Hence, the Monitoring Service first monitors the SLA Client Responsibilities of the hosting SLA. If clients send a higher number of requests than that allowed, clients are violating the SLA. No corrective actions are performed to reconfigure the cluster in this case; rather, an application level exception is raised that may cause the misbehaving clients to be put in a position not to interfere with the properly behaving ones. Second, the Monitoring Service monitors the Server Responsibilities of the hosting SLA. If it detects that the cluster SLA violation rate trend is close to breaching the hosting SLA, it invokes the Configuration Service so as to reconfigure the cluster. In this case, the Configuration Service acts upon the cluster by adding new nodes up to a predefined limit. That limit is a configuration parameter obtainable via either application benchmarking or application modeling. Its purpose is to identify an upper boundary above which adding new nodes does not introduce further significant performance enhancements. This can be caused by factors such as increased coordination costs for cluster management and bottlenecks due to shared resources such as a centralized load balancing service or a centralized DBMS.

Note that the Configuration Service can augment the cluster by introducing one new node at a time or more than one in a single action. When adding one node at a time, a waiting time elapses between the Configuration Service reconfigurations following each node inclusion. This time may be useful for handling the transient phase of a new added node. The transient phase represents the time elapsed from the introduction of the new node in the cluster until it reaches a steady state enabling it to serve the client requests. On the other hand, adding more than one node at a time can be useful to deal with possible flash crowd events. In fact, these events may not be fully resolved by adding just one node at the time to the cluster, owing to the above-mentioned transient phase.

If the Monitoring Service detects that the cluster is effectively responding to the injected client load, it invokes the Configuration Service to act upon the cluster by releasing clustered nodes, as they are no longer necessary. In configuring/reconfiguring the cluster, the Configuration Service produces a resource plan object. This object includes the IP address of each clustered node belonging to the built cluster configuration. In essence, the resource plan specifies the resources to be used in order to construct the QoS-aware cluster capable of meeting the input hosting SLA.

## 4.A CASE STUDY: THE ENHANCED JBOSS APPLICATION SERVER

JBoss consists of a collection of middleware services for communication, persistence, transactions, and security [18]. These services interact by means of a microkernel based on the Java Management eXtension (JMX) specifications [29].

Fig.3 shows how the QoS-aware cluster is implemented with a number of clustered QaAS nodes.

This figure shows that every clustered node incorporates a replica of the Configuration Service, Monitoring Service, and Load Balancing Service, each implemented and integrated into the JBoss application server as an MBean. Only one node in the cluster is responsible for SLA enforcement, monitoring, and load balancing. We term this node the cluster Leader. The remaining nodes, called slave nodes, are used as backup servers in case the Leader crashes.

Possible Leader crash during configuration (or runtime reconfiguration) is detected by the Configuration Services in the slave nodes through their (local) Monitoring Services. These Monitoring Services are alerted of the Leader’s crash by the underlying group communication mechanism, namely, JGroups [24], included in the standard JBoss application server. JGroups [2] provides the clustered nodes with reliability properties that include lossless message transmission, message ordering, and atomicity. As a result, should Leader crash occur, the following simple recovery protocol is performed by the Configuration Service instances deployed in the slave nodes. Every Configuration Service is identified by a unique identifier (ID) consisting of the IP address of the machine where the Configuration Service is deployed. In addition, all Configuration Services have a consistent cluster configuration state object; this is the resource plan object mentioned earlier and consists of a list of the IDs of the available clustered nodes. When Leader crash is detected by the slave Monitoring Services, the latter inform their local Configuration Services that a new Leader must be elected. The Configuration Services examine the IDs of the available nodes in the cluster configuration state and elect the server with the minimum ID as the new Leader. Note that, owing to the JGroups reliability properties mentioned earlier, all clustered nodes have a consistent view of the current cluster membership; hence, they can easily apply the simple deterministic algorithm for Leader election introduced above.

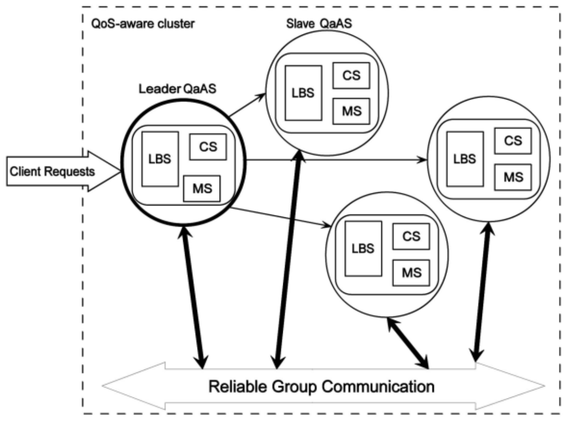

**Fig 3 QOS aware application server**

The first election of the cluster Leader is triggered by the hosting SLA deployment. In fact, the QaAS node where that deployment occurs becomes the Leader. The Configuration Service in the Leader node parses the input hosting SLA to extract the QoS parameters that guide the required cluster configuration (client requestRate, serviceAvailability, efficiency); it then makes them available to the Monitoring Service responsible for checking cluster performance. For this purpose, the Monitoring Service is constructed out of three components: SLA Violations Monitor, Evaluation and Violation Detection Service, and Cluster Performance Monitor.

In general, these components interact with each other to implement a monitoring mechanism capable of dynamically adapting to modifications of both the client load characterization and node operational conditions. In our implementation, we assume that node performance degradation can be due to the load imposed by other services running on the nodes (nodes can concurrently host and run services other than QaAS).

The above-mentioned Monitoring components are invoked when incoming client requests are intercepted by the Load Balancing Service. These requests are intercepted by a LoadBalancingFilter implemented using the Servlet Filter technology [17]. The main responsibilities of the Monitoring components can be summarized as follows: The SLA Violations Monitor is responsible for verifying whether or not the SLA efficiency attribute is met within the SLA efficiency validity period. When violations of the hosting SLA occur 4.1 **4.1 Experimental Evaluation**

The prototype described above has been used to carry out a set of experiments aimed at assessing 1) the overhead introduced by our middleware services in the JBoss application server, 2) the scalability properties of our QoSaware cluster, and 3) the resource optimization achievable in a QoS-aware cluster, while honoring the hosting SLA.

In a test of several Linux machines interconnected by a dedicated 1 Gb Ethernet LAN. Each machine is a 2.66 Ghz Intel Xeon processor, equipped with 2 GB RAM. In the experiments described below, one of these machines is dedicated to host the cluster Leader; the other machines are used to host either the QaAS slave nodes serving the client requests or the client program used to generate artificial load in the cluster. In addition, a dual-processor machine is dedicated to hosting the database used in the experimental evaluation, namely MySQL [34].

As for the client program, we implemented our own program in order to 1) specify a variety of client load distributions, 2) specify different client request rates, and 3) simulate typical behavior of common browsers by enabling caching of the static contents of the HTTP client requests.

### 4.1.1 QaAS Overhead Evaluation

First concern was to assess whether our middleware services were adding unnecessary overhead to the cluster response time and throughput, in the absence of failures. For this purpose, we instantiated the middleware services in the cluster introduced earlier and used from one up to four QaAS nodes. With these configurations, we ran two sets of tests. In the first set, we directly injected equally distributed artificial client requests to each a vailable standard JBoss node. In the second set of tests, we deployed the hosting SLA, thereby enabling our services and directed the client requests to the Load Balancing Service.

In both cases, the cluster provided the same throughput and response time, showing that QaAS does not introduce any significant overhead.

Note that introducing a reverse proxy implies performance penalties; however, these are balanced by the HTTP protocol optimizations performed by the Load Balancing Service. Similar results can be obtained with advanced HTTP reverse proxies such as Apache HTTP server with mod_jk [32].

To conclude this section, we measured the saturation point of the Load Balancing Service. For this purpose, we used the in-memory database [19] replicated in each clustered nodes and then through the Load Balancing Service until we were able to identify the maximum load above which the Load Balancing Service becomes a bottleneck. From this test, we observed that the Load Balancing Service was capable of supporting up to 450 requests per second introducing no overhead. Note that this figure depends principally on the Web page size rather than the number of nodes used in the cluster.

### 4.1.2 QaAS Scalability Evaluation

The second experiment was conducted to evaluate the scalability of the QoS-aware cluster we had developed. In this experiment, we varied the number of nodes in the cluster starting by one node, scaling up to four nodes. The obtained results are shown in Table 1. It is clear that, by augmenting the number of QaAS clustered nodes, QaAS does scale, even if not in an entirely linear fashion. In fact, as evident in Table 1, for two nodes, throughput is exactly double compared to the value obtained with one node. With three and four nodes, throughput keeps on augmenting, although not linearly. We identified the cause of this behavior in the database, which becomes a bottleneck. Note that the Load Balancing Service could not have caused these performance anomalies, as throughput is below the 450 requests per second mentioned in the previous section.

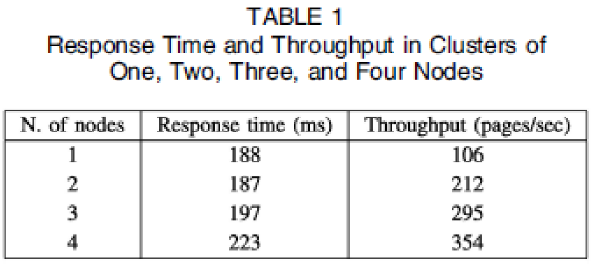

TABLE 1
Response Time and Throughput in Clusters of One, Two, Three, and Four Nodes

| N. of nodes | Response time (ms) | Throughput (pages/sec) |
|---|---|---|
| 1 | 188 | 106 |
| 2 | 187 | 212 |
| 3 | 197 | 295 |
| 4 | 223 | 354 |

### 4.1.3 Resource Utilization Evaluation

The purpose of this final experiment was to assess the ability of our middleware to optimize clustered nodes utilization without causing hosting SLA violations. In carrying it out, we assumed that the absence of dynamic clustering techniques (such as those enabled by QaAS) means a resource overprovision policy is used. This statically allocates as many nodes as possible to ensure honoring the hosting SLA. The maximum number of nodes available was fixed to four. Therefore, in an over-provision policy, all four nodes are used; in contrast, to honor the bookshop hosting SLA, our middleware allowed us to dynamically allocate a minimum of one up to four clustered QaAS nodes depending on the imposed load at different time intervals.

For the purposes of this experiment, nodes were made available in a pool of spare nodes ready to be included in the cluster as required. cluster following a simple request distribution: Our program client gradually raised bookshop application HTTP request rate up to 360 requests per second; the load then gradually decreased to 2 requests per second. The bold line in Fig. 4 shows this distribution. It follows that, if no QaAS is being used, the standard JBoss clustering approach has to allocate all four available nodes and maintain them allocated to the bookshop application for the entire duration of the test, regardless of the actual client load. In other words, it needs resource overprovision (see the lighter area in Fig. 4), which guarantees the hosting SLA is met. In contrast, QaAS dynamically adjusts the cluster size as necessary, augmenting the number of clustered nodes as load increases and releasing nodes as load decreases, as illustrated by the darker area in Fig. 4. In conclusion, to offset SLA violations, the QaAS trend in resizing the cluster follows the distribution of the imposed load, as shown in Fig. 4 (yet again, the darker area mentioned above). In this test, we also measured the percentage of SLA violations (see Fig. 5). Here, the peaks correspond to the instant in which a new node had to be added to the cluster for not incurring SLA efficiency requirement violations; however, as can be seen in Fig. 4, the SLA violation rate is maintained below the limit imposed by the hosting SLA.

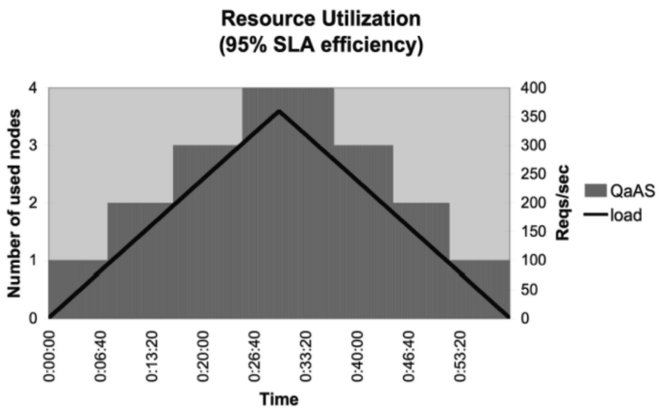

**Fig 4. Resource Utilization**

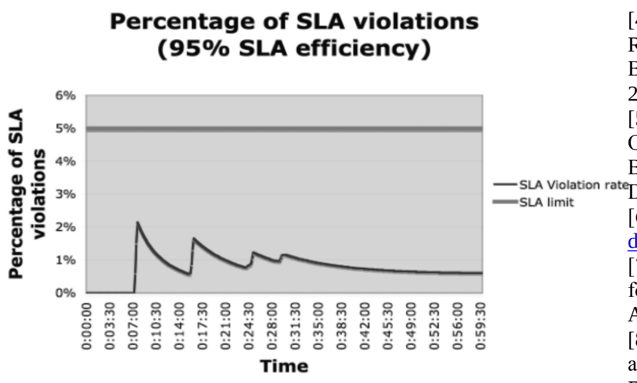

**Fig 5.SLA Violation**

**Conclusion**

In our architecture, the size of the cluster can change at runtime, in order to meet nonfunctional application requirements specified within what we have termed a hosting SLA.

The experimental results we have presented show the effectiveness of our approach; in particular, they show that the efficient use of resources and the strict constraints imposed by the SLA can be addressed by means of dynamic reconfiguration mechanisms even in the case of such complex systems as a cluster of J2EE application servers.

We are investigating issues of dynamic resource management when multiple applications are concurrently deployed in a J2EE server cluster; these applications have their own hosting SLAs and compete for the use of the same clustered nodes.

**REFERENCES**

[1] "Service Level Agreement (SLA)," http://www. wilsonmar.com/ 1websvcs.htm, 2006.
[2] T. Abdellatif, E. Cecchet, and R. Lachaize, "Evaluation of a Group Communication Middleware for Clustered J2EE Application
Servers," Proc. Int'l Symp. Distributed Objects and Applications (DOA '04), Oct. 2004.
[3] K. Appleby, S. Fakhouri, L. Fong, G. Goldszmidt, M. Kalantar, S. Krishnakumar, D.P. Pazel, J. Pershing, and B. Rockwerger, "Oceano-SLA Based Management of a Computing Utility," Proc. Seventh IFIP/IEEE Int'l Symp. Integrated Network Management (IM)
May 2001.
[4] M. Aron, P. Druschel, and W. Zwaenepoel, "Cluster Reserve: A Mechanism for Resource Management in Cluster-Based Network," Proc. ACM SIGMETRICS Conf., June 2000.
[5] J. Balasubramanian, D.C. Schmidt, L. Dowdy, and O. Othman, "Evaluating the Performance of Middleware Load Balancing Strategies," Proc. Eighth Int'l IEEE Enterprise Distributed Object Computing Conf. (EDOC '04), Sept. 2004.
[6] "WebLogic Clustering," BEA Systems, http://e-docs.bea.com/ wls/docs81/cluster/, 2006.
[7] "BEA WebLogic Server 8.1 Overview: The Foundation for Enterprise Application Infrastructure," BEA Systems, Aug. 2003.
[8] S. Bouchenak, F. Boyer, E. Cecchet, S. Jean, A. Schmitt, and J.B. Stefani, "A Component-Based Approach to
Distributed System Management—A Use Case with Self-Manageable J2EE Clusters," Proc. 11th ACM SIGOPS European Workshop, Sept. 2004.
[9] M.J. Buco, R.N. Chang, L.Z. Luan, C. Ward, J.L. Wolf, and P.S. Yu, "Utility Computing SLA Management Based Upon Business Objectives," IBM Systems J., 2004.
[10] ObjectWeb home page, ObjectWeb Consortium, http://www. objectweb.org, 2006.
[11] G. Coulouris, J. Dollimore, and T. Kindberg, Distributed Systems —Concepts and Design, fourth ed. Addison-Wesley, 2005.
[12] M. Debusmann and A. Keller, "SLA-Driven Management of Distributed Systems Using the Common Information Model," Proc. Eighth Int'l IFIP/IEEE Symp. Integrated Management (IM), Mar. 2003.
[13] "SPECjAppServer2004," Standard Performance Evaluation Corp., http://www.spec.org/jAppServer2004, 2006.
[14] B. Roehm et al., IBM WebSphere Application Server V6 Scalability and Performance Handbook. Redbooks IBM Corp., 2004.
[15] B. Roehm et al., IBM WebSphere V5.1 Performance, Scalability, and High Availability WebSphere Handbook Series. Redbooks IBM Corp., 2004.
[16] P. Asadzadeh et al., "Global Grids and Software Toolkits: A Study of Four Grid Middleware Technologies," High-Performance Computing— Paradigm and Infrastructure, 2006.
[17] Servlet filter, http://java.sun.com/products/servlet/Filters. html, 2006.
[18] M. Fleury and F. Reverbel, "The JBoss Extensible Server," Proc.
ACM/IFIP/USENIX Int'l Middleware Conf., June 2003.
[19] HSQLDB, http://www.hsqldb.org/, 2006.
[20] The WebSphere Application Server, IBM, http://www-306.ibm. com/software/webserver/appserv, 2006.
[21] "Utility Computing," IBM Systems J., vol. 43, 2004.

---******------