SLA-Driven Clustering of QoS-Aware Application Servers

Giorgia Lodi, Fabio Panzieri, Davide Rossi, and Elisa Turrini

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.

1 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) [43] servers (e.g., [20], [7], [23], [10]), CORBA Component Model (CCM) [48] servers, or .NET [39]. 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) [38]. 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.

• The authors are with the Department of Computer Science, University of Bologna, Mura Anteo Zamboni 7, 40126 Bologna, Italy.
E-mail: {lodig, panzieri, rossi, turrini}@cs.unibo.it.

Manuscript received 2 Feb. 2006; revised 27 July 2006; accepted 5 Dec. 2006; published online 26 Jan. 2007.
Recommended for acceptance by W. Emmerich.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-0024-0206.

The design of a QoS-aware application server requires a careful assessment of the correct amount (and characteristics) of the resources needed to meet application SLAs. In general, applications hosted in application servers (e.g., Web applications, Web services) are characterized by high load variance; hence, the amount of resources needed to honor their SLAs may vary notably over time. In order to ensure that an application SLA is not violated, one can adopt a *resource overprovision* policy, based on evaluating (either through application modeling or through application benchmarking) all possible resources a hosted application could require in the worst case and then statically allocating these resources to the application. This policy can lead to a largely suboptimal utilization of the hosting environment resources; being based on a worst-case scenario, a number of allocated resources could remain unused at runtime. In contrast, optimal resource utilization can be achieved by providing (and maintaining at runtime) each hosted application with the minimum number of resources required to meet the application SLA.

In view of the above observations, 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*. The relative responsibilities of these three services are discussed in detail later in this paper.

The next section presents the context in which the software solution under proposal, and the technologies commonly deployed for implementing and hosting distributed enterprise applications, may be used.

1.1 Scenario

We are currently witnessing a trend in the IT market toward the adoption of *on-demand computing models*, which allow service providers to make computational resources available to customers when needed.

Utility computing is one service provision model that can be classified as an on-demand computing model. Utility computing is an emerging industrial practice often defined as "development in IT outsourcing, whereby service capacity is provided as needed and the customers pay only for the actual use" [21]. Its main goal is to maximize the efficient use of the resources while minimizing the enterprise costs. In other words, the utility computing model supports a service provision that maintains the minimum amount of computational resources necessary to meet application QoS requirements.

In this context, resources can be acquired dynamically at runtime by considering both the runtime conditions of the computing environment and the gathering of the application QoS requirements that are to be honored.

In the enterprise environment, component-based technologies, such as J2EE [43], CCM [48] or .NET [39] application servers are often used to host distributed enterprise applications. In this paper, we concentrate on the J2EE framework and the support it provides to application developers for the development of J2EE-based applications. A J2EE-based application consists of managed components, namely, servlets and Enterprise Java Beans (EJBs), hosted in the servlet container and the EJB container, respectively. These containers are the runtime environments for the application components and provide them with a federated view of a collection of middleware services for security, persistence, transaction, life-cycle management, and so on. In addition, clustering is supported by most J2EE commercial and open-source application servers, for scopes of scalability, high availability, and resilience.

Clustering is a well-known architectural mechanism that can be used to improve both throughput and availability of a distributed application via load balancing and replication. In the J2EE context, a cluster can be constructed out of multiple application server instances that cooperate in order to host distributed J2EE applications. In a clustered environment, containers are distributed, and each node runs an instance of them.

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.

In view of the above observations, we have developed a software architecture that allows dynamic J2EE application server clustering and automatic cluster reconfiguration at runtime. Our architecture allows a J2EE cluster to react to possible changes of its own operational conditions that could result in violations of application QoS requirements.

The architecture we have developed springs from an earlier one designed in the context of the TAPAS EU-funded project (IST Project No. IST-2001-34069) [40]. In order to assess the effectiveness of our middleware architecture in honoring SLAs, we implemented a prototype using the open-source, J2EE application server JBoss [23]. Here, we describe our latest design and implementation and discuss the testing and evaluation.

The paper is structured as follows: The next section introduces the concept of SLA and describes a specific example used in our development. Section 3 discusses the principal issues addressed in the design of the Configuration Service, Monitoring Service, and Load Balancing Service mentioned earlier. Section 4 describes our middleware architecture prototype and examines its experimental evaluation. Section 5 compares and contrasts our approach with relevant related work. Finally, Section 6 features our conclusions.

2 Service Level Agreements

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

To date, no conventional standard definition of SLA exists. Relevant research is underway to investigate the definition of languages for SLA specification (e.g., [45], [26]). Though the results of this research fall outside the scope of this paper, we used a format inspired by SLAng [45] for defining an SLA in the implementation of our software architecture.

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 (used in our tests, as discussed later). 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>
```

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 the example above, the daily availability of the bookshop application is to be no less than 99 percent).

In addition, each application operation specified as part of the SLA Server Responsibilities can be classified 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. The `efficiency` and `efficiencyValidity` attributes capture this SLA requirement. In particular, in the example above, these attributes specify that no less than 95 percent of client requests are to be served within any two-hour interval—in compliance with the SLA Server Responsibilities requirements. Hence, we do assume that the SLA QoS attributes (such as response time) can be violated during the SLA efficiency validity period (two hours in the above example), provided that the violation rate is maintained below the SLA efficiency limit (in our example, the percentage of tolerated violations is 5 percent).

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. The services were developed to minimize their interdependency with specific J2EE implementations and maximize the portability of our software architecture.

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 (see Section 4) computed and updated at run time. These parameters allow the 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 session-based load balancing, client requests belonging to a specific client session are dispatched to the same clustered node (this client-server correspondence is termed *session affinity*).

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. For this purpose, the policy considers the number of pending requests in the Load Balancing Service queues associated with each clustered node; so, when an incoming client request is intercepted by the Load Balancing Service, the request is forwarded to the clustered node with the shortest pending requests queue. This policy allows us to dynamically evaluate the expected performance of each clustered node, even in the presence of load imposed by other services running on the nodes.

We have compared, contrasted, and evaluated different static and adaptive load balancing policies; in all, we favor the use of our WorkLoad policy. For the sake of conciseness, a report of this assessment is not contained in this paper; interested readers can refer to [27].

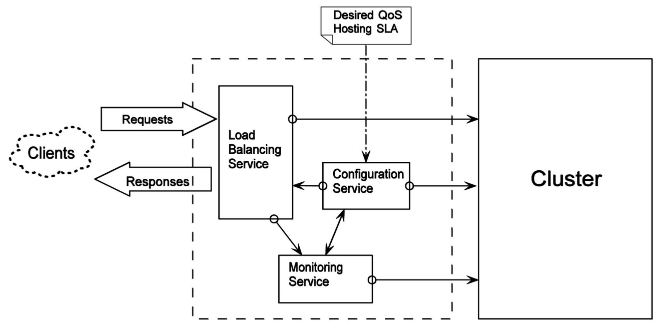

Fig. 1. QoS-aware middleware services interaction.

3.1 QoS-Aware Middleware Services Interactions

Our QoS-aware middleware services cooperate with each other to ensure hosting SLA enforcement and monitoring.

Fig. 1 shows how they interact. In Fig. 1, 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 (that is, the violation rate trend of the cluster is significantly below the hosting SLA limit), 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.

Node failures and voluntary connections to the QoS-aware cluster are detected by the Monitoring Service, which then raises an exception to the Configuration Service. In both cases, the Configuration Service reconfigures the cluster; that is, it updates the resource plan by removing (or adding) the node(s) that have become unavailable (or available); in addition, in case of node failures it adds new nodes should the modified cluster configuration be incapable of meeting the hosting SLA.

Once the Configuration Service has produced the resource plan, it transmits it to the Load Balancing Service, as depicted in Fig. 1, to enable the Service to dispatch the incoming client requests toward the new set of clustered nodes. The Load Balancing Service does not need to be statically configured with the membership of the cluster; rather, it becomes aware of it at runtime, in line with Configuration Service reconfigurations.

Finally, the architecture described in this paper should be applicable, with minor adaptations, to any application servers that support some form of "middleware extensions" and allow such extensions to interact with the application server's cluster manager. Currently, this set of application servers includes, for example, JBoss [23] and JOnAS [10].

4 A CASE STUDY: THE ENHANCED JBOSS APPLICATION SERVER

We implemented a prototype of our middleware architecture as an extension of the JBoss v. 4.0.4.GA [23] 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]. JMX defines a common software bus that allows Java developers to integrate components such as modules, containers, and plug-ins. These components are declared as Managed Bean (MBean) services; they are loaded into JBoss and administered by the JMX software bus.

A number of JBoss application server instances can be clustered in a local area network. A JBoss cluster consists of a set of nodes, where a node is a JBoss application server

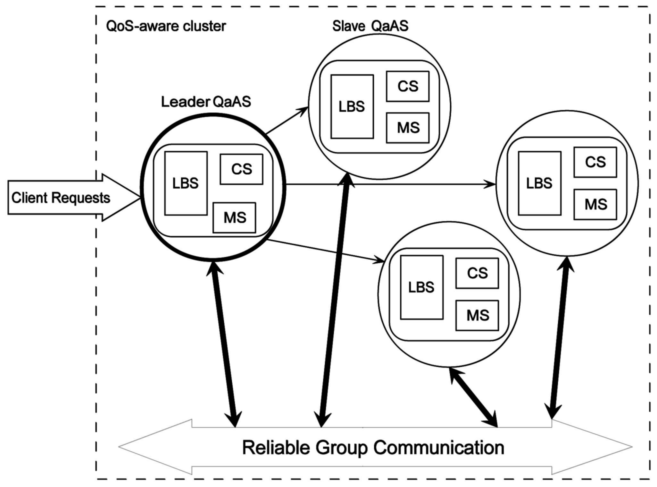

Fig 2. QoS-aware cluster.

instance. To simplify our discussion, we assume below that each clustered node is configured with a static IP address; hence, each node runs in a dedicated clustered machine and no DHCP server is used to dynamically assign the IP address to the machines in a JBoss cluster. Our prototype (termed *QaAS*) is a standard JBoss server enhanced with our middleware architecture incorporating the three services described above.

Fig. 2 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.

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 that may result in (economic) penalties, the Monitoring Service raises an exception to the application level.

The Evaluation and Violation Detection Service and the Cluster Performance Monitor cooperate with each other in checking, for each HTTP request, whether cluster response time meets that specified in the hosting SLA, and triggering the cluster reconfiguration, if necessary. The decision to trigger cluster reconfiguration is based on a specific warning threshold and a collection of indices maintained by the Cluster Performance Monitor.

For the sake of conciseness, in this paper, we have not described all these indices and thresholds (interested readers should refer to [27]); rather, we focus on the most relevant: the efficiency index.

The efficiency index allows the Monitoring Service to check whether or not the current violation rate trend, exhibited by the cluster and related to the current cluster load conditions, may lead to SLA efficiency violations during the validity period. This index is a negative integer initialized to 0, which can be incremented or decremented, during a fixed monitoring interval, depending on the SLA response time violations.

Specifically, let $\mathcal{E}$ be the SLA efficiency; the efficiency index is decremented by $(1/(1-\mathcal{E}))-1$, provided the actual request response time violates the related SLA `maxResponseTime`. As long as its value is not equal to 0, it is incremented by 1 if the actual request response time does not violate the related SLA requirement.

To clarify, let us consider the example SLA in Section 2. This SLA states that the response time QoS requirements can be violated at most 5 percent of the time, i.e., $\mathcal{E}=0.95$, in any two-hour interval (as specified by the `efficiencyValidity` attribute). Thus, for each incoming client request, if the cluster does not meet the SLA response time for such a request, the efficiency index is decremented by $(1/(1-\mathcal{E}))-1$, i.e., 19 in this case (otherwise, it is incremented by 1). This behavior guarantees that if the index ranges within $[-((1/(1-\mathcal{E}))-1),0]$ ($[-19,0]$ in this case), the current violation rate trend remains within the SLA efficiency. In contrast, if the index falls below $-((1/(1-\mathcal{E}))-1)$ (-19, in this example), the current violation trend of the cluster might lead to a violation of the SLA efficiency.

In the latter case, the Monitoring Service may require a cluster reconfiguration that adds new clustered nodes. To this end, the efficiency index is periodically compared to a dynamic threshold that depends on the SLA violations trend. In particular, if the efficiency index exceeds the threshold, reconfiguration is necessary to adapt the cluster to the new load conditions. As stated above, other indices and thresholds are maintained by the Monitoring Service. These are used to decide if a node can be released from the cluster by estimating the cluster performance trend without that node [27].

The Configuration Service reconfiguration that adds (or releases) clustered nodes has been implemented by using two main techniques. The first technique consists of checking if a pool of spare nodes is available; if it is, the spare nodes are QaAS nodes ready to be used to serve client requests. In contrast, in the second technique, the Configuration Service starts up (or shuts down) QaAS nodes. To facilitate this, we implemented an external program (that is, a program not integrated in the JBoss application server) to be invoked to run (or shut down) QaAS nodes when necessary. This external program can be also used to acquire on-demand resources owned by third-party organizations. Note that, as the use of this program is time-consuming (there is the start-up time to take into account when a node is added to the cluster), the Monitoring parameters can be properly configured to allow QaAS to apply a more conservative node-releasing policy, and a more timely node-adding policy, thus preventing SLA violations.

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 QoS-aware cluster, and 3) the resource optimization achievable in a QoS-aware cluster, while honoring the hosting SLA.

In our tests, we used a cluster configuration consisting 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. Note that existing performance measurement tools such as ECperf [13] or JMeter [25] do not provide us with all these features.

The clustered servers homogeneously host a digital bookshop application we developed (issues of homogeneous and heterogeneous deployment are discussed in [28]). This provides clients with the operations mentioned in Section 2. The bookshop application is a J2EE application consisting of both Web and EJB components. The application was constructed out of Java Server Page (JSP), Servlet, Stateless Session Beans, and Entity Beans. We used Entity Beans Container Managed Persistence (CMP) to represent the application's persistent objects mapped to the database.

Finally, in all experiments, we used a per-session load balancing approach, disabling the HTTP session replication on the clustered nodes serving the clients requests [42], [41]. With this setup, if a node crashes while serving a client session, the session cannot be recovered at the application server level. Consequently, either the client program implements its own recovery mechanisms, or this program may have to start the session again. This behavior is reasonable for most Web applications; if this is not the case, HTTP session replication must be enabled.

4.1.1 QaAS Overhead Evaluation

Our 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 available 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 node to avoid it becoming a bottleneck. Hence, we first injected artificial load directly onto the 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.

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

4.1.2 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.

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.

In view of these assumptions, this final experiment consisted of three principal tests. In the first, we set the SLA efficiency attribute to 95 percent; this meant the most an SLA response time requirement could be violated was 5 percent of the time over a predefined timeframe. This test ran for approximately one hour; the results obtained are illustrated in Fig. 3. We injected artificial load into the

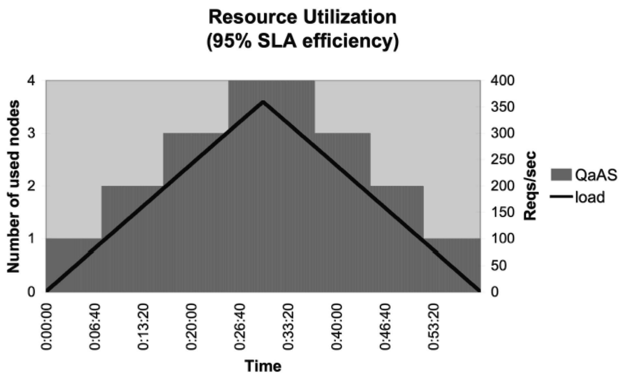

Fig. 3. Resource utilization.

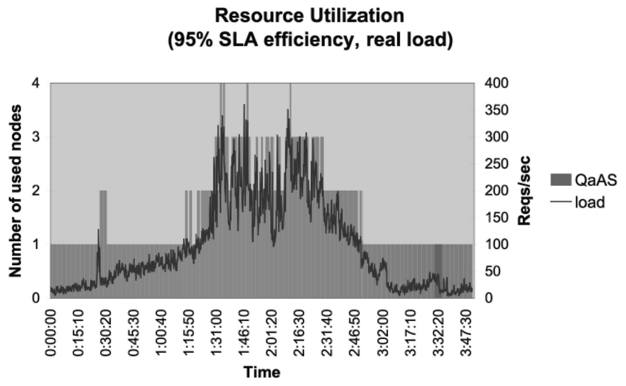

Fig. 5. Resource utilization: real load.

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. 3 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. 3), 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. 3. In conclusion, to offset SLA violations, the QaAS trend in resizing the cluster follows the distribution of the imposed load, as shown in Fig. 3 (yet again, the darker area mentioned above). In this test, we also measured the percentage of SLA violations (see Fig. 4). 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.

In the second test, we used a different load distribution. This distribution was derived by analyzing logs from Bologna University's portal access. These logs contained HTTP request rates relating to a period of approximately 24 hours. We augmented the rate frequency by a factor of six compared to the original, thus obtaining a load characterization distributed over 4 hours. This load distribution modification allowed us to have highly variable request rates thus submitting the cluster to critical load conditions. The results obtained by this test are shown in Fig. 5. Here, it is evident that the load distribution presents peaks which QaAS handles with an increased number of used nodes. Even in this case, QaAS used clustered nodes that follow the load distribution. The percentage of SLA violations in this test are shown in Fig. 6, which reveals a higher SLA violation rate compared to that observed in the first test. This is chiefly caused by the critical conditions as described above. However, here again, the violation rate is maintained below the SLA limit, thus allowing SLA honoring.

In the final test, we used the same load distribution as in the first test and we observed the behavior of our middleware architecture in the presence of node crashes. Therefore, we shut down a QaAS node at runtime, obtaining the results shown in Fig. 7. As depicted in Fig. 7, when QaAS was using two clustered nodes, it detected the node crash and added a new node in a timely manner. This behavior corresponds to the peak in the QaAS resource utilization graph in Fig. 7.

The percentage of SLA violations in this test are illustrated in Fig. 8. In contrast to the first test, the number of SLA response time violations augmented in correspondence to the node crash up to 4 percent (owing to the missing crashed node). Afterward, the SLA violations

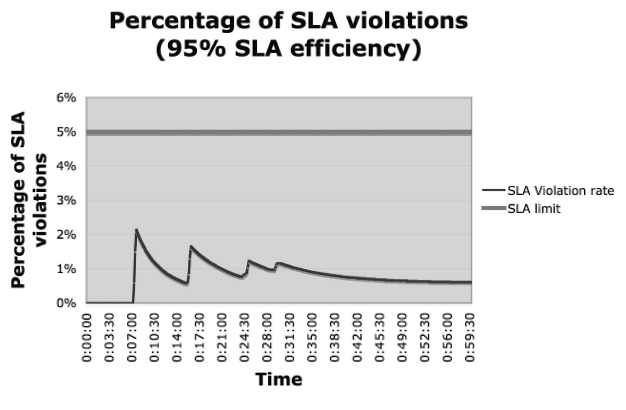

Fig. 4. SLA violations.

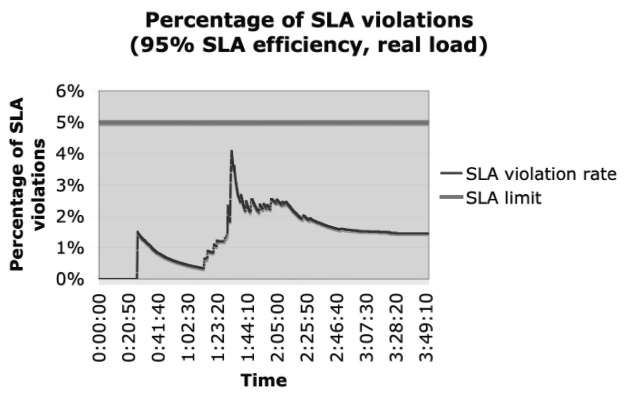

Fig. 6. SLA violations: real load.

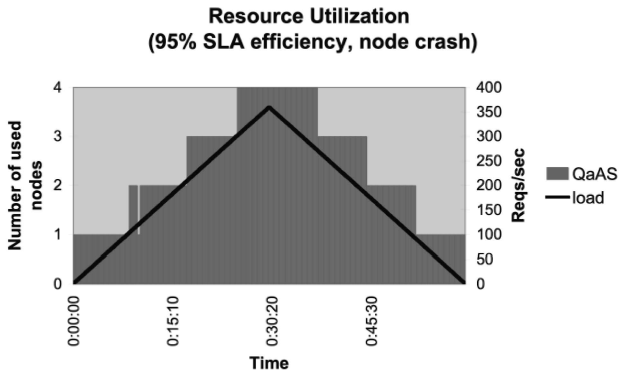

Fig. 7. Resource utilization: node crash.

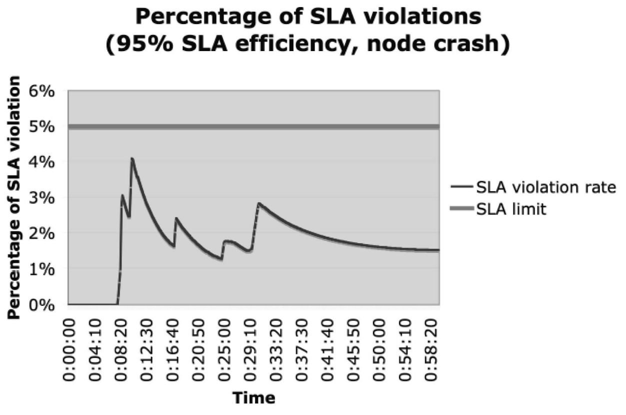

Fig. 8. SLA violations: node crash.

decreased until the end of the test. Note that, even in this case, our middleware architecture did not exceed the 5 percent SLA efficiency limit; however, this architecture is not designed to face an arbitrary number of failures.

5 Related Work

Recent years have witnessed considerable research into QoS enforcement and monitoring in the context of Web Services [22], [37] and application server technologies (see below) with a series of relevant proposals that have notably influenced our approach to QaAS design. Here, we compare and contrast our approach to these proposals.

De Miguel's work [30] outlines new models of QoS-aware components and QoS-aware containers. It proposes an enhancement of such containers with facilities for QoS configuration and negotiation. This approach differs from our approach in that it involves new types of application components and a new runtime environment for them in order to meet nonfunctional application requirements. We, however, have not created new types of application components, choosing to integrate our middleware services with those already provided by application server technologies.

Resource clustering issues have been widely investigated in the literature. For example, [47] describes techniques for the provision of CPU and network resources in shared hosting platforms (platforms constructed out of clusters of servers), running potentially antagonistic third-party applications. The architecture proposed in [47] provides applications with performance guarantees by overbooking clustered resources and combining this technique with commonly used resource allocation mechanisms. This approach is similar to that discussed in [4], which describes a framework for resource management in Web servers, for delivering predictable QoS and differentiated services. However, in [4], application overload is detected when resource usage exceeds certain predetermined thresholds, while, in [47], detection takes place by observing the tail of recent resource usage distributions. Shen et al. [44] investigate the design and implementation of an integrated resource management framework for cluster-based network services. An adaptive multiqueue scheduling scheme is employed inside each node of a clustered environment to achieve both efficient resource utilization under quality constraints and provide service differentiation. Nonetheless, although these approaches present techniques for optimizing clustered resources, they differ from our approach in that they operate at different levels of abstraction and do not use the concept of SLA to specify QoS requirements.

SLA enforcement and monitoring are discussed in [26], [12]. They 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, [50] presents an architecture for the coordinated enforcement of resource sharing agreements (SLAs) among applications using clustered resources. The design approach here shares a number of similarities with our approach (for example, resource sharing among applications is governed by the SLAs these applications have with their hosting environment). However, prototype implementations of this architecture were developed at both the HTTP level and transport level, rather than at the middleware level, as in our approach.

Another project with notable influence on ours is the IBM Oceano project [3]. Oceano is a prototype of a highly available, scalable, and manageable infrastructure for an e-business computing utility. It manages the resources of the computing utility in an automatic and dynamic manner by reassigning resources to distributed applications in order to meet specific SLAs. Oceano includes SLA-driven monitoring, event correlation, network topology discovery, and automatic network reconfiguration. Though its approach is quite similar to ours, to the best of our knowledge, its prototype is not specifically applied to an application server context, a move we propose in this paper. In addition, Oceano's load balancing takes place at the network level rather than the middleware level.

In the J2EE context, several research projects examine dynamic clustering management issues. In [8], the authors discuss the design of a modular architecture to build command and control loops to manage complex distributed systems. Here, the architecture builds self-manageable J2EE application server clusters to simplify the administration of such distributed systems and enable dynamic reconfiguration capabilities. These capabilities are effective both in case of failures of J2EE components and in case of increased cluster load. Nevertheless, to the best of our knowledge, this work does not focus on providing dynamic clustering techniques to honor SLAs, as we propose, and it monitors low level resources such as CPU, memory, and disk I/O usage rather than application level QoS requirements.

In [36], the authors present a platform that can manage complex, multitiered applications. The platform is implemented as an extension of the WebSphere application server and each request is served by multiple resources, distributed over multiple tiers. Here, service providers are allowed to divide all requests from different Web applications into service classes, according to a user-defined policy; in addition, each class is assigned a performance goal. This describes a target response time and includes an importance level to determine the relative importance of meeting, exceeding, or violating target SLA requirements. These performance goals and importance targets serve to allocate the necessary resources to the application requests. This IBM research focuses on the same issues discussed in our paper. However, in [36], the application server maps observed performance to a continuous utility function constantly adapting the hosting environment to maximize that utility function; in our approach, however, QaASs use a warning threshold and set of indexes to adapt the execution environment to the changes occurring at runtime. An implementation of the architecture proposed in [36], termed WebSphere eXtended Deployment (XD), is currently available by acquiring a commercial license. This product is a specialized software application, available for z/OS only.

In essence, our assessment of current commercial J2EE application servers reveals that, with the exception of WebSphere, they do not offer mechanisms for SLA-driven dynamic resource management. Third-party companies (e.g., Data Synapse [46]) offer extensions for commercial and open-source J2EE application servers that could be similar to our own; however, to date, no details are available concerning such commercial products that allow us to compare and contrast the performance of these products with that we obtained from our implementation.

As for the load balancing activity, in [5], the authors present an adaptive load balancing service implemented using standard CORBA features. The key features of this load balancing service, named Cygnus, are 1) it is able to make load balancing decisions based on application-defined load metrics, 2) it dynamically (re)configures load balancing strategies at runtime, and 3) it transparently adds load balancing support to client and server applications. In contrast, our approach is applied to a component-based technology such as J2EE; hence, we now focus on load balancing services enabled in such an environment.

In a J2EE context, widely used proprietary application servers such as WebSphere and WebLogic provide users with HTTP load balancing mechanisms similar to our solution. The WebLogic platform consists of a WebLogic Server equipped with a set of Web server plug-ins. These plug-ins are modules that can be added to third-party Web Servers and configured appropriately to enable interactions between the WebLogic Server and the application components hosted inside proprietary Web Servers. In general, the plug-ins are used to distribute the HTTP client requests among the WebLogic clustered servers; their static load balancing policies are 1) *Round Robin*, which cycles through the list of WebLogic Server instances, and 2) *Weight-Based*, which improves the previous Round Robin algorithm by taking into account a preassigned weight for each server [6]. No adaptive policies are provided. Furthermore, these plug-ins use a first list of target WebLogic Server instances as the starting point for load balancing among clustered members. After the first request is routed, a dynamic list of servers is returned. This mechanism allows WebLogic to provide users with a dynamic load balancing technique in terms of variations in the cluster membership configuration (our approach also provides this feature).

In the WebSphere application server, load balancing can take place on two different levels. In fact, the first solution proposed by WebSphere [15] includes the load balancing mechanism inside a plug-in for the Web container. The plug-in distributes the HTTP client requests based on two different algorithms, namely, the *Round Robin with Weighting* algorithm, which cycles through the list of WebSphere instances and decrements their weight by 1 when a server is first selected, and the *Random* algorithm. Note that, in contrast to our WorkLoad policy, these two load balancing strategies are static and unable to handle variations in the computational load of the servers. In addition, the list of available WebSphere instances is not dynamically determined, as in both our and WebLogic's case; it is included in a specific configuration file. The second solution proposed by WebSphere [14] uses an IBM software packet named *WebSphere Edge Components*, featuring a *Load Balancer*. This Load Balancer consists of different components. The most important is the *Dispatcher* component. The Dispatcher distributes the load it receives to a set of servers contained in the cluster, and then decides which server handles the HTTP request, based on the weight of each server in that cluster. The weight can be set as a fixed value (unchangeable, regardless of the conditions of the balanced servers) or dynamically computed by the Dispatcher. In this latter case, the list of available clustered servers is obtained dynamically. In conclusion, our approach is similar to the second WebSphere solution.

6 Concluding Remarks

This paper has discussed the design, implementation, and experimental evaluation of a J2EE-based dynamic clustering architecture, constructed out of multiple, interconnected QoS-aware application servers.

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.

Given this context, we have designed and developed a set of services that optimize the resource usage of a J2EE application, hosted in our clustered architecture, to allow the application hosting SLA to be honored without incurring in resource overprovision costs.

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.

As an extension of the work described in this paper, 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.

The design principles and solutions we have adopted in the development of our QoS-aware clustering mechanisms may well be deployed in emerging application areas, such as the Service Oriented Architecture (SOA) and Grid computing areas.

As observed in [31], service providers will be offering commercial services in SOAs, based on SLAs; in addition, Grid provides an ideal framework for SOAs, as it enables the management of collections of resources. It follows that an SLA-driven approach to the dynamic allocation and management of Grid resources, such as that described in this paper, may well be incorporated as part of the Grid architecture's core middleware services [31], in order to avoid the use of static, resource over-provision policies.

Therefore, our future work will include an investigation into the use of our clustering mechanisms in these contexts and in application servers distributed across wide area networks. Finally, in order to augment the robustness of our architecture, we will investigate issues of application server fault-tolerance, in general, and survivability to multiple server crashes, in particular.

Acknowledgments

The authors wish to thank their colleagues involved in the TAPAS project, and the TAPAS project Advisory Board Members, for their comments on earlier versions of this paper. In addition, they thank the Bologna University portal staff for making available to them the data used in their tests, and the anonymous reviewers for their invaluable comments and suggestions.

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.
[22] D.B. Ingham, S.K. Shrivastava, and F. Panzieri, "Constructing Dependable Web Services," *IEEE Internet Computing,* vol. 41, no. 1, Jan.-Feb. 2000.
[23] "JBoss Enterprise Middleware System," JBoss, http://www.jboss.org, 2006.
[24] "JGroups—A Toolkit for Reliable Multicast Communication," http://www.jgroups.org, 2006.
[25] JMeter, http://jakarta.apache.org/jmeter/, 2006.
[26] A. Keller and H. Ludwig, "The WSLA Framework: Specifying and Monitoring Service Level Agreements for Web Services," Research Report RC22456, IBM, Mar. 2003.
[27] G. Lodi, "Middleware Services for Dynamic Clustering of Application Servers," Technical Report UBLCS-2006-06, PhD thesis, Univ. of Bologna, Apr. 2006.
[28] G. Lodi and F. Panzieri, "QoS-Aware Clustering of Application Servers," *Proc. First IEEE Int'l Workshop Quality of Service in Application Servers/23rd Int'l Symp. Reliable Distributed Systems (SRDS '04),* Oct. 2004.
[29] "Java Management Extension Instrumentation and Agent Specification v.1.1," Sun Microsystems, http://java.sun.com/jmx, 2002.
[30] M.A. De Miguel, "QoS-Aware Component Frameworks," *Proc. 10th Int'l Workshop Quality of Service (IWQoS '02)* 2002.
[31] B. Mitchell and P. Mckee, "SLAs a Key Commercial Tool" *Innovation and the Knowledge Economy: Issues, Applications, Case Studies,* 2005.
[32] Apache Mod_jk, http://jakarta.apache.org/tomcat/connectors-doc/, 2006.
[33] J. Myerson, "Use SLAs in a Web Services Context, Part 1: Guarantee Your Web Service with a SLA," http://www-128.ibm.com/developerworks/webservices/library/ws-sla/, Oct. 2004.
[34] MySQL, http://www.mysql.com, 2006.
[35] K. Ogata, *Modern Control Engineering,* third ed. Prentice Hall, 1997.
[36] G. Pacifici, W. Segmuller, M. Spreitzer, M. Steinder, A. Tantawi, and A. Youssef, "Managing the Response Time for Multi-Tiered Web Applications," IBM Research Report RC23651, IBM, 2005.
[37] G. Pacifici, M. Spreitzer, A. Tantawi, and A. Youssef, "Performance Management for Web Services," Research RC22676, IBM, Aug., 2002.
[38] "SLA for Application Service Provisioning," ASP Industry Consortium White Papers, http://www.allaboutasp.org, 2006.
[39] J. Proise, *Programming Microsoft .NET, Core Reference,* Microsoft Press, 2002.
[40] "Trusted and QoS-Aware Provision of Application Services," IST Project No. IST-2001-34069, TAPAS Project, http://tapas.sourceforge.net, 2006.

[41] D. Rossi and E. Turrini, "Testing J2EE Clustering Performance and What We Found There," *Proc. First IEEE Int'l Workshop Quality of Service in Application Servers,* Oct. 2004.

[42] D. Rossi and E. Turrini, "Analyzing the Impact of Components Replication in High Available J2EE Clusters," *Proc. Joint Int'l Conf. Autonomic and Autonomous Systems and Int'l Conf. Networking and Services,* Oct. 2005.

[43] B. Shannon, *Java 2 Platform Enterprise Editon v. 1.4—Final Release,* 24 Nov. 2003.

[44] K. Shen, H. Tang, T. Yang, and L. Chu, "Integrated Resource Management for Cluster-Based Internet Services," *Proc. Fifth Symp. Operating Systems and Design and Implementation,* Dec. 2002.

[45] J. Skene, D. Lamanna, and W. Emmerich, "Precise Service Level Agreements," *Proc. 26th Int'l Conf. Software Eng. (ICSE '04),* 25 May 2004.

[46] Data Synapse, http://www.datasynapse.com, 2006.

[47] B. Urgaonkar, P. Shenoy, and T. Roscoe, "Resource Overbooking and Application Profiling in Shared Hosting Platforms," *Proc. Fifth Symp. Operating Systems and Design and Implementation,* Dec. 2002.

[48] N. Wang, D.C. Schmidt, and C. O'Ryan, "An Overview of the CORBA Component Model," *Computer-Based Software Engineering,* Addison-Wesley, 2000.

[49] E. Wustenhoff, *Service Level Agreement in the Data Center.* Sun BluePrints, Apr. 2002.

[50] T. Zhao and V. Karamcheti, "Enforcing Resource Sharing Agreements among Distributed Server Clusters" *Proc. 16th Int'l Parallel and Distributed Processing Symp. (IPDPS)* Apr. 2002.

Giorgia Lodi received the "Laurea" and PhD degrees in computer science from the University of Bologna, Italy. She was a research associate of computer science at the University of Newcastle upon Tyne, United Kingdom, in 2002. She is currently a research associate of computer science at the University of Bologna, Italy. Her research interests include distributed systems, middleware, application server technologies, clustering techniques, and SLA management.

Fabio Panzieri received the "Laurea" degree in "Scienze dell' Informazione" from the University of Pisa, Italy, and the PhD degree in computer science from the University of Newcastle upon Tyne, United Kingdom. He was appointed a professor of computer science at the University of Bologna, Italy, in 1990. His research interests span many areas of distributed computing, including fault tolerance, real-time, and middleware and communication support for large-scale distributed applications.

Davide Rossi received the "Laurea" degree in "Scienze dell'Informazione" and the PhD degree in computer science from the University of Bologna, Italy. He is an assistant professor in the Department of Computer Science at the University of Bologna, Italy. His research interests include coordination models and systems, distributed systems/middleware architectures, software engineering, Web engineering, workflow languages and systems, and business process management.

Elisa Turrini received the "Laurea" and PhD degrees in computer science from the University of Bologna, Italy, where she is currently a research associate at the Department of Computer Science. Her research interests include distributed systems, system performance evaluation, coordination languages, and workflow systems.