TAPAS

IST-2001-34069

Trusted and QoS-Aware Provision of Application Services

TAPAS Architecture
QoS Enabled Application Servers

Report Version: Deliverable D7

Report Delivery Date: 31 March 2003

Classification: Draft – for comments only

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

Project Co-ordinator: Newcastle University

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

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

TAPAS Architecture

QoS Enabled Application Servers

Giovanna Ferrari[1], Giorgia Lodi[2]
(F. Panzieri[2], S. K. Shrivastava[1], Eds.)

Table of contents:

---

[1] Department of Computing Science, University of Newcastle upon Tyne, Newcastle upon Tyne, NE1 7RU (UK).

[2] Dipartimento di Scienze dell’Informazione, University of Bologna, Mura Anteo Zamboni 7, 40127 Bologna (Italy).

4th IEEE Int. Workshop on Policies for Distributed Systems and Networks, Como (Italy), June 4-6, 2003.

Introduction

This Report describes the architecture of the TAPAS middleware platform that will be developed as part of the TAPAS project. This architecture consists of a collection of middleware services that extend the abstraction of application server, as provided by current middleware technologies, such as the CORBA Component Model and Java 2 Enterprise Edition (J2EE), in order to meet Quality of Service (QoS) application requirements, such as performance, reliability, and security. The motivations for this architecture can be summarized as follows.

Current component-oriented technologies allow designers to construct distributed applications out of reusable and interoperable software components (e.g., commercial operating systems, communication protocols, middleware services). These technologies support the specification of the functional component interfaces; however, they support only partially the definition of non-functional properties (i.e., the QoS) of the component execution.

These technologies promote the use of *containers* to host application component instances. Specifically, a container provides the run time environment for those instances, and shields them from the complexity of most of the system services, such as the transaction, security, persistence, and notification services. Hence, containers take part in the management of the non-functional properties of the components they host.

Several containers can be hosted by the same *application server*; thus, QoS negotiation, establishment, and adaptation facilities can be added to the application server and used by component containers to make them QoS-aware.

These facilities are implemented in the TAPAS middleware architecture by two principal middleware services, named *Configuration Service* and *Controller Service*, respectively, that can be used to extend an application server. The former service is responsible for discovering, negotiating, and reserving the resources necessary to meet the QoS requirements of a particular application component, hosted by that application server; the latter service is responsible for monitoring the reserved resources, and possibly adapting the component execution in case the QoS delivered by these resources deviates from that required by the component itself.

The TAPAS architecture described in this Report uses Service Level Agreements (SLAs), as discussed in the TAPAS deliverable D1 [Beckman *et al.* 2002], in order to derive the QoS application component requirements, and monitor the delivered QoS at the component run time. Thus, SLAs, in the TAPAS architecture, are used not only as an inter-organizational contractual feature, but also to govern the component execution.

In order to define the TAPAS architecture, we have carried out an extensive assessment of the current state of the art in the design of architectures developed to meet QoS requirements of distributed applications. From this assessment it has emerged that none of the architectures we have examined fully meet the TAPAS objectives; however, this assessment has allowed us to

derive a number of recommendations and design principles we have deployed in the definition of our architecture. These include both such recommendations as the need for incorporating a resource monitoring service in our architecture, in order to assess the resource state at run time, as well as design principles such as those that can be derived from the control theory [Ferrari G. 2002] in order to deploy adaptation facilities.

Further, we have examined two use cases (namely, the hosting of a generic application by an Application Service Provider, and that of specific auction application) in order to expose specific requirements the TAPAS architecture has to meet. As a result of this activity, we have identified a detailed set of functionalities our architecture is to incorporate in order to meet its objectives.

We have then studied how to integrate these functionalities in existing component-based middleware platforms. Specifically, we have examined the integration of these functionalities within the J2EE application server, and their instantiation within two specific implementations of the J2EE platform; namely, Jboss [JBOSS 2003] and JOnAS [JONAS 2003]. In addition, in this context, we have examined how the proposed TAPAS architecture would enable the deployment of a specific replication technology, such as that described in [Morgan *et al.*, 2002] (attached to this Report).

This Report is structured as follows. In the next Section we discuss our state of the art assessment of QoS architectures. In Section 3 we introduce the two use cases mentioned above, and discuss the functionalities the architecture we propose incorporates. Section 4 describes how these functionalities can be integrated in the J2EE platform, and discusses the deployment of the replication technology described in [Morgan *et al.*, 2002]. Finally, Section 5 concludes this Report.

2 End-to-end QoS Architectures

QoS has been defined as "A set of quality requirements on the collective behaviour of one or more objects" [ITU/ISO 1995]. Specifically, in the distributed multimedia context, it has been defined as "The set of those quantitative and qualitative characteristics of a distributed multimedia system that are necessary in order to achieve the required functionality of an application" [Vogel *et al.* 1995].

At the low levels of a distributed system architecture, QoS refers to the ability of the network to deliver the most appropriate communication service that can deal effectively with specific data traffic requirements (e.g., constant bit rate, variable bit rate). Thus, issues of QoS have been addressed principally in the design of communication protocols and mechanisms that allow the programmer to control such communication parameters as network throughput, packet delay, delay jitter, and packet loss (e.g. RSVP [Braden *et al.* 1997], IntServ [Braden *et al.* 1994], DiffServ [Carson *et al.* 1998]) over QoS-enabled communication technologies, such as ATM (Asynchronous Transfer Model) and SONET (Synchronous Optical Network) [Lodi 2002].

These parameters indeed affect the user-perceived QoS of a distributed application; however, further QoS requirements emerge at the application level, as pointed out in [Ferrari 1998], which are generally related the non functional properties of the applications themselves. Specifically, these may include performance oriented requirements, such as timeliness of execution and relative processing speed; reliability oriented requirements, such as high-availability and failure

recovery; security oriented requirements, such as authentication, privacy, anonymity and confidentiality. As these requirements pertain to the distributed application components (and the end-systems hosting these components) at the end points of a communication subsystem, they are termed “end-to-end QoS requirements”.

Note that these requirements cannot be *fully* met at the communication subsystem level, as they fall outside the scope and responsibility of this level (rather, this level provides support which can be crucial in order to meet them). For example, the implementation of a highly available distributed filing service can typically require that both reliable communications be provided at the communication subsystem level, and appropriate service replication techniques be deployed at the application level, as these replication techniques concern strictly the distributed filing service architecture using the communication subsystem.

Owing to the above observation, end-to-end QoS can be thought of as a pervasive system property, as depicted in Figure 1 below, which is to be preserved by orchestrating carefully the cooperation among both the distributed application components (and their hosting end-systems), and the communication subsystem.

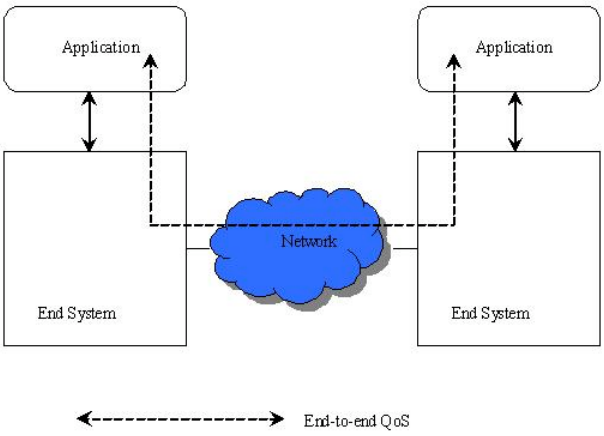

Fig. 1: End-to-End QoS

In general, the QoS required from a given system can be expressed as a collection of <parameter-value> pairs. Each parameter can be considered as a typed variable whose value can range in a given set. In a large scale distributed context, such as that enabled by the current Internet technology, QoS communication parameters apply to low level network protocols, as mentioned above, and may not be under the control of the application.

In one such context, lack of control of these parameters may exacerbate the complexity of meeting QoS application requirements. Specifically, over the Internet, this complexity arises principally from the IP-based *best effort* communication service model, currently available over this network. Namely, within this model: (i) applications do not have the ability to reserve bandwidth, and (ii) there is no means to exercise control over the network resources (i.e., no admission control policy is implemented).

This model is indeed adequate for traditional applications such as FTP and e-mail, as these applications are not influenced notably by the delays that the network can introduce in delivering the information; however, it is inadequate for those distributed applications in which the *perceived latency* (i.e., the response time) dominates the application performance. These include soft real-time applications, such as IP telephony applications, as well as interactive distributed applications such as multi-party games over the Internet, or applications implementing Web access to multimedia data (note that these latter applications may not have to meet hard real-time requirements, although they typically exhibit specific latency requirements which may range between a minimum and a maximum level [Ghini 2002]).

In particular, in an Internet based distributed scenario, the availability of the communication resources may dynamically change during the execution of an application; hence, in order to maintain the expected QoS levels, that application requires that monitoring and adaptation mechanisms be available, so as to cope with possible fluctuations of resource availability.

In order to address these issues, in recent years a large body of research has investigated the design and development of so-called end-to-end QoS architectures, aimed at providing platforms that support effectively distributed applications characterized by end-to-end QoS requirements.

Relevant examples of these architectures include the QoS Broker [Nahrstedt & Smith 1995], RT CORBA [Fay *et al.* 1997]), TAO [Schmidt *et al.* 1997], QuO [Zinky *et al.* 1997], Agilos [Baochun 2000], the Real-Time Specification for Java (RTSJ) [Bollella & Gosling 2000, ControlWare [Zhang *et al.* 2002], the AMIDST [Bergmans *et al.* 2002], the Policy Based System [Kakadia 2000], and, finally, the QoS Controller architecture for e-commerce sites [Menasce' *et al.* 2001]. In the following Subsections we examine these architectures in detail.

2.1 The QoS Broker

The QoS Broker [Nahrstedt & Smith 1995] has been developed in order to support distributed multimedia applications (DMMAs) in synchronous (i.e. characterized by predictable delays) communication environments (e.g. ATM networks). Its principal responsibility consists of configuring the communication subsystem in order to transmit flows of multimedia data in a guaranteed manner.

To this end, the QoS broker (i) cooperates with both the local OS and the communication subsystem in order to achieve a balance among DMMA QoS requirements, local OS resources, and network resources, and (ii) negotiates the use of remote resources with the remote QoS brokers. The QoS broker can be implemented as an extension of the communication subsystem software, as illustrated in Figure 2, below.

It is structured in two principal components; namely, the broker buyer and the broker seller. The broker buyer orchestrates local resources, and gathers information about network and remote resources from the sellers. The seller waits for requests form buyers, and negotiates both network parameters, and local resources.

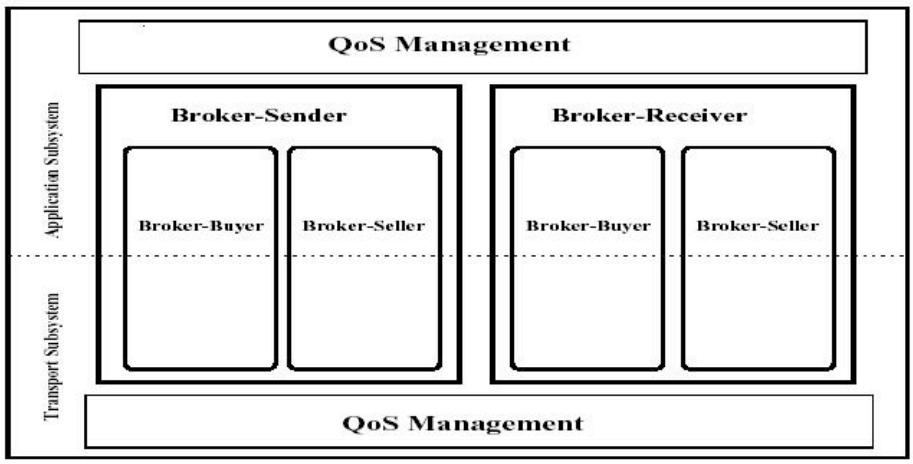

Fig. 2: QoS Broker Architecture

2.2 Real-Time CORBA

RT CORBA is a standard architecture for real-time management of distributed objects; this architecture has been developed in order to support fixed-priority CORBA applications [Fay *et al*. 1997].

RT CORBA prescribes a number of functionalities that must be integrated and managed by ORB end-systems in order to ensure the predictable behavior of the activities carried out by CORBA clients and servers [Schmidt & Kuhns 2000].

The above mentioned functionalities include:

- *Communication infrastructure resource management*: a RT CORBA end-system must exploit policies and mechanisms in the underlying communication infrastructure that support resource guarantees.
- *OS scheduling mechanisms*: the RT-CORBA specification is targeted to operating systems that allow applications to specify scheduling priorities and policies.
- *Real-Time ORB end-system*: a real-time ORB end system must provide applications with standard interfaces that allow these applications to specify their resource requirements.
- *Real-Time services and applications*: Real-Time CORBA ORBs must guarantee efficient, scalable and predictable behavior of higher-level services and application components.

In order to manage these functionalities, RT CORBA defines standard interfaces and QoS policies that improve an application's ability to configure and control (i) the processor resources through pools of threads, priority mechanisms, intra-process mutual exclusion mechanisms and a global scheduling service, (ii) the communication resources via protocol properties and explicit bindings of clients to servers and (iii) the memory resources via request queues and bounded thread pools.

In essence, RT-CORBA specifies a set of relevant features that allow the application to control thread priorities and scheduling; however, it does not provide high level primitives for constructing adaptive QoS management mechanisms.

2.2.1 TAO

TAO is an open-source high performance RT-CORBA-compliant ORB. TAO implements a rich set of middleware mechanisms that can support applications with both deterministic and statistical QoS requirements, and applications with best-effort requirements.

This ORB consists of the following four major components that provide applications with end-to-end QoS guarantees:

- *ORB*: this component supports real-time by optimizing (i) code generation, and (ii) the use of system components such as the memory management system, and the network protocols.
- *Scheduling Service*: this service provides the applications with real-time scheduling of client requests, and supports both static scheduling, based on off-line schedulability analysis, and dynamic scheduling, via admission control policies.
- *Event Service*: this service implements real-time scheduling of CORBA events, and provides filtering and correlation mechanisms that allow consumers to select the events they receive.
- *Real-Time I/O (RIO) subsystem*: this subsystem runs in the OS kernel and is designed to take advantage of ATM network features.

It is worth observing that programming TAO's low level real-time mechanisms, in order to meet specific end-to-end QoS requirements, can be complex and error-prone, particularly for large-scale, QoS-enabled distributed applications. Therefore, higher-level middleware capabilities for end-to-end QoS specification and control are required. In order to meet these requirements, a complementary architectural framework, called Quality Objects (QuO), has been developed. This framework is introduced below.

2.2.2 QuO

QuO is a framework designed to support the development of distributed applications characterized by QoS requirements [Loyall *et al.* 1998]. It provides the application designer with the ability to i) specify, monitor, and control QoS aspects of a distributed object application, and ii) define and implement application adaptation mechanisms that must be enabled in response to changing system conditions (e.g., an intrusion aware application detecting an attack to a server object may decide to break the connection to that server, locate a server that has not been attacked, and reconfigure itself to use that latter server).

In the QuO framework, a method call made by a client on a remote object, through its functional interface, is a superset of a traditional CORBA call. This superset includes the additional components itemized below, and depicted in Figure 3:

- *Contracts* between clients and objects: the *Contract* component specifies the level of service required by a client, the level of service a remote object is expected to provide its clients with, the operating regions indicating the possible measured QoS, and actions to take when the QoS level changes.

- *Delegates* of remote objects: a *Delegate* component is a wrapper of a remote object, and provides a functional interface identical to that of the remote object it wraps; in addition, this component implements locally adaptive behavior, based upon the current state of the system QoS. For each method invocation, the *Delegate* is responsible for reading the QoS contract.

- *System condition objects*: these components implement interfaces to resources, mechanisms, objects and ORBs in the system that require to be measured and controlled by QuO *contracts*.

Moreover, the QuO toolkit includes:

- a *Quality Description Language (QDL)*, used for describing contracts, system condition objects, and the adaptive behavior of the objects and the delegates;

- the *QuO kernel*, which coordinates the evaluation of contracts and the monitoring of the system condition objects;

- *Code Generators*, which weave together the QDL descriptions, the QuO kernel code, and the client code to produce a single program.

When the client invokes a method on the remote object, it is actually invoking that method on the local delegate, which triggers contract evaluation. The contract component gets the actual values of the system conditions, to determine the current operating region. The delegate chooses the behavior based on the current regions; for instance, the delegate might i) chose between alternative methods, or ii) block when QoS has degraded, or iii) pass the method invocation through to the remote object. Then the remote object is invoked, performs its method and returns a value. By this time, the delegate performs similar processing upon the method return, i.e. it evaluates the contract to obtain the QoS regions and selects a behavior, passing then the return value back to the client [Vanegas *et al.*1998].

In summary, QuO provides a middleware platform capable of supporting end-to-end QoS. Its important features are the QuO *Contracts*, and the *QDL* implemented to describe these *Contracts*. However, QuO monitoring and adaptation mechanisms do not appear to be sufficiently flexible, as they can be activated only when an invocation on a remote method is made by a client, or the result of that invocation is returned to that client.

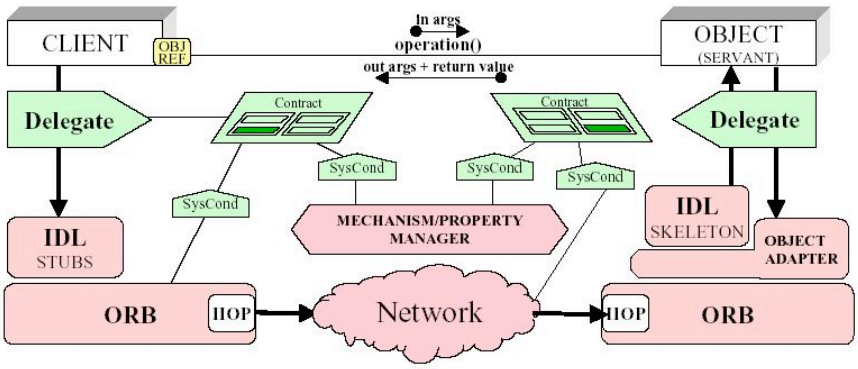

Fig. 3: QuO Framework

2.3 Agilos

Agilos (Agile QoS) is a middleware architecture designed to provide services that can support so-called *application-aware* QoS adaptation mechanisms for use from distributed applications structured as a set of CORBA objects.

The Agilos adaptation mechanisms are fully configurable to the application's needs, and *aware* of the application-specific semantics. These mechanisms monitor the system resources, and maintain system-wide adaptation properties of the applications. As they are implemented at the middleware level, they do not require tight integration or modifications to the services implemented in the OS kernel and network protocol stack.

Agilos is designed as a three-tier architecture, as illustrated in Figure 4. The first tier embodies two components, namely the *adaptors* and the *observers*, which maintain tight relationships with individual resources, and support low level resource adaptation by reacting to changes in the availability of those resources.

The second tier consists of application-specific *configurators*, responsible for taking decisions as to when and what application functions are to be invoked in a client-server application (based upon on-the-fly user preferences and application-specific rules). Moreover, this tier includes so-called *QualProbes* components that provide QoS probing and profiling services, so that application-specific adaptation rules can be either derived by measurements, or specified explicitly by the user.

The third tier, on both clients and servers, consists of a centralized *gateway* and multiple *negotiators* that control the adaptation behavior of an application constructed out of multiple clients and servers, so that dynamic reconfigurations of the client-server mapping are possible, and can be tuned to the application.

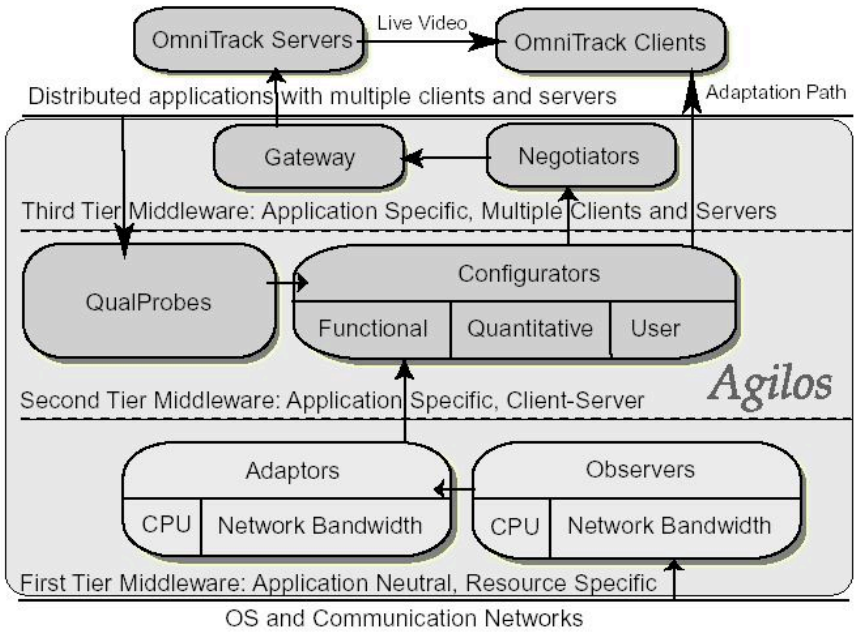

Fig 4: Agilos Architecture

It is worth observing that both the middleware components and the actual QoS-aware applications may be reconfigured to adapt to the changing environment. Thus, as pointed out in [Baochun 2000], the following two distinct approaches exist to the design of *application-aware* QoS adaptation mechanisms.

One approach, adopted by the *QuO* middleware described earlier, is to dynamically reconfigure the middleware itself so that it can provide a stable and predictable operating environment to the application, transparently to the application itself. This approach is attractive as it does not require any modifications to the application; hence, any legacy application can be deployed with little efforts and with a certain level of QoS assurance. However, since it can only provide a generic solution to all applications, a set of highly application-specific requirements cannot be addressed.

In contrast, the middleware can be *active*, and exert strict control of the adaptation behavior of the QoS-aware applications, so that these applications adapt and reconfigure themselves under such control. This approach has the advantage of knowing exactly what are the application-specific adaptation priorities and requirements, so that appropriate adaptation choices can be made to address these requirements. The *Agilos* middleware architecture implements this approach.

2.4 Real-Time Specification for Java

The Real-Time Specification for Java (RTSJ) extends the Java programming language and the Java Virtual Machine specifications in order to provide an Application Programming Interface that enables the creation, verification, analysis, execution and management of real-time Java threads (i.e., Java threads whose correctness criteria include timeliness) [Bollella *et al.* 2000].

The Real-Time for Java Experts Group identified the following seven areas of interest that required Java language extensions; namely, thread scheduling, memory management, thread synchronization, asynchronous event handling, asynchronous control transfer, asynchronous thread termination, and physical memory management.

Thus, in essence, RTSJ consists of a set of primitives that extend the Java programming language so as allow the programmer of real-time Java applications to deal with the seven areas of interest mentioned above.

2.5 ControlWare

ControlWare is a middleware platform developed to provide Internet services with performance guarantees. This platform applies methods, derived from the Control Theory, for system configuration and control purposes. Specifically, using ControlWare, the controlled system can be an Internet service, and the control goal is to provide that service with QoS guarantees.

ControlWare provides the Internet service developer with software tools and library routines for converting QoS specifications (i.e., the required QoS guarantees) into so-called *feedback control loops*; these *loops* are stored in service configuration files, and implement the service performance control mechanisms.

In addition, ControlWare implements a convenient interface between the service software and the feedback control loops mentioned above. This interface, illustrated in Figure 5, is termed *SoftBus*, and is responsible for the management of the monitoring of the controlled system, and its adaptation to possible variations of the execution environment. The *SoftBus* is a distributed protocol running across multiple machines and address spaces, forming a virtual application backbone into which applications, performance sensors, and actuators can plug-in. Sensors and actuators measure the system performance, and implement the required adaptation strategies.

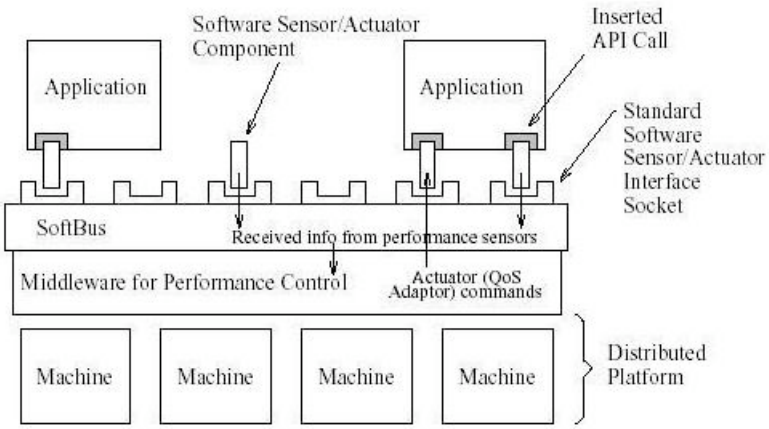

Fig. 5: ControlWare Architecture

2.6 The AMIDST Project

This project investigates the design of a specialized middleware framework that meets QoS application requirements. Similar to ControlWare, this framework deploys mechanisms derived from the Control Theory in order to i) control that the QoS application requirements be

effectively met, and ii) adapt the execution environment to possible variations in the resource availability which may occur.

As illustrated in Figure 6, the AMIDST Project middleware framework consists of a *middleware platform* and a *control loop*. The middleware platform incorporates the computing and communication resources which may have to be manipulated in order to meet some *agreed QoS* levels (i.e., the QoS application requirements). The *control loop* is responsible for assessing whether or not the *agreed QoS* is effectively met, and adapting the execution environment, if necessary.

The platform maintains *probes* that enable *sensors*, in the control loop, to observe the QoS delivered by the computing and communication resources. An *interpreter* evaluates the QoS observed by a sensor, according to some metric relevant to the specific QoS parameter under observation.

For example, in a client-server application, it can be required that the client perceived response time fall, on average, within some agreed QoS range. To this end, it can be necessary firstly to observe and sample a number of *client request transmission*, and relative *server response delivery*, times; secondly, to calculate the average response time obtained from those observations. In this example, this calculation is carried out by the above mentioned interpreter.

A *comparator* compares the value of a particular QoS parameter, returned by the interpreter, with the *agreed QoS* value of that parameter. If a difference between these two values is detected, a *decider* is enabled, which selects an appropriate *control strategy*, consisting of objectives to be reached in the execution of the control loop.

A *translator* is responsible for translating the control strategy into a collection of *control actions*, (i.e., manipulations of the controlled system); finally, an *actuator* schedules these control actions so that they are carried out using one or more probes.

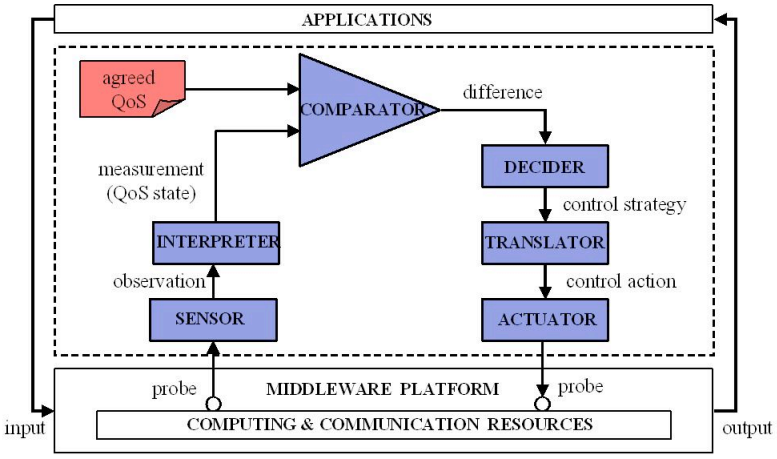

Fig. 6: AMIDST Architecture

2.7 Policy Based System

A QoS Policy Architecture has been proposed by Sun Microsystems [Kakadia 2000] in the area of so-called Policy Based Systems. These are systems that control the access to the

networking and computing resources using policies that reflect business goals in an automated manner. Figure 7 shows the architectural components of the Sun Microsystems Policy Based Architecture.

This Architecture relies on the concept of *policy*; i.e., a high level abstraction that represents a business objective. Typically, a policy can be specified in a variety of "formats" (e.g., a natural language), and requires a translation from that language into simple device-specific configurations.

In general, the process of Policy based administration starts with a Service Level Agreement (SLA) between a provider and a customer; that SLA can be translated into a policy object representing the policy abstraction. A policy object is specified and entered in the Policy Management Tool (PMT) that validates it, resolves possible conflicts, stores it in persistent storage, and forwards it to the Policy Decision Point (PDP). The PDP reads a policy object, translates it into device-specific configurations, and forwards those configurations to the appropriate Policy Enforcement Points (PEPs). The PEPs must ensure that the policies are carried out by allocating appropriate resources, such as CPU, bandwidth, and buffer spaces; to this end, the PEPs reconfigure the devices, based on the configurations provided by the PDP.

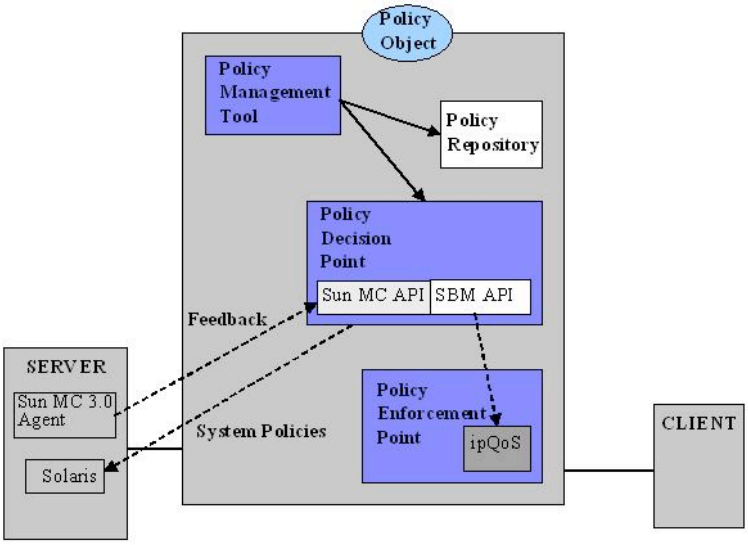

Fig. 7: Policy Architecture

2.8 QoS Controller

The QoS Controller [Menasce' *et al*. 2001] has been developed in order to monitor and adapt e-commerce sites so that desired QoS levels can be attained. The QoS Controller monitors the e-commerce site, and uses control theory-based techniques to determine the values of various configuration parameters. These parameters can be changed dynamically in order to ensure that the site shows as little deviation as possible from the desired QoS levels. In essence, assuming that an e-commerce site deploys the QoS Controller, this Controller periodically collects data, in time intervals named Controller Intervals (CIs); at the end of each CI, the QoS Controller decides whether or not reconfiguration has to take place.

The principal components of the QoS Controller are the Workload Monitor (WM), the Performance Monitor (PM), the Configuration Controller (CC), the Performance Model Solver (PMS), and the QoS Monitor (QoSM), illustrated in Figure 8.

The WM collects information about the arrival rate of requests, and computes the average request arrival rate in a CI. The PM measures device (e.g. CPU, disks) utilisations for the Web applications and database server machines running at the e-commerce site. Using this information the PM computes the service demand (i.e. the total service time per request at a given time) at each device during the CI.

The QoSM checks, at the end of each CI, if any of the QoS metric has been violated by receiving information of completing requests from the e-commerce site; this component decides whether there is a need to change the site configuration. In the affirmative case, it instructs the CC to determine a new configuration for the site.

Finally, the PMS computes the QoS values for each configuration it receives from the CC, using service demands received from the PM and the arrival rate received from WM.

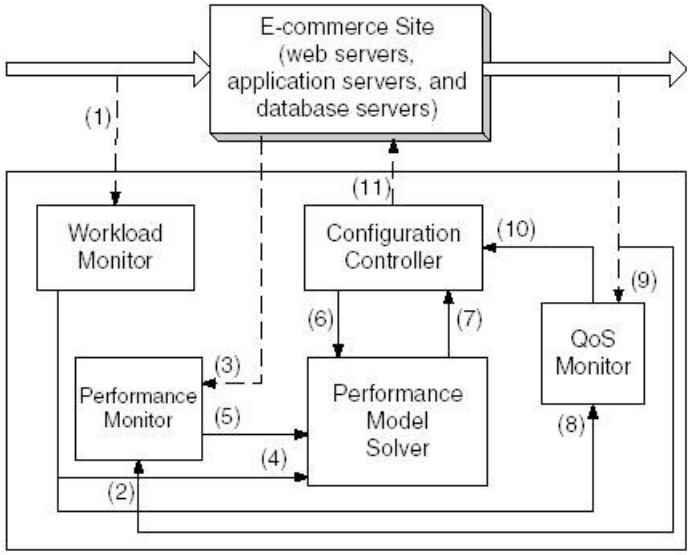

Fig. 8: QoS Controller Architecture

To conclude our state of the art assessment, it is worth observing that, in essence, the architectures we have examined consist of a collection of QoS-aware middleware services providing end-to-end QoS guarantees. These services are typically responsible for meeting the non-functional application requirements mentioned above, so as to allow the application developer to concentrate on the implementation of the functional aspects of his/her application.

The distinguishing features of these architectures can be summarized as follows. The QoS Broker, RT Corba, TAO, and RTSJ architectures address issues of end-to-end QoS in multimedia and mission critical distributed applications, incorporating resource reservation and monitoring mechanisms. QuO embodies adaptation mechanisms that can be enabled at the time a method is invoked. Agilos provides the applications with support for the invocation of

adaptation mechanisms implemented at the application level. Finally, ControlWare, AMIDST, Policy Based System, and the QoS Controller incorporate adaptation mechanisms whose internal logic is based on the feedback control theory.

In the following Section we examine two specific use cases that allow us to derive the principal end-to-end QoS requirements the TAPAS architecture has to meet, and to identify the relevant services this architecture is to incorporate in order to meet those requirements. The motivations for a use cases driven approach to requirements capturing are summarized below.

3 Architectural design issues

Use cases have become a common practice for capturing the functional requirements of a computing system. They originated from the object-oriented community; however, their applicability is not limited to object-oriented systems, only. The use cases driven approach has been introduced by Ivar Jacobson in [Jacobson 1987, Jacobson 1992], and is based on describing usage scenarios of a *system under development* (*sud*, in the following). Specifically, a use case is not a single scenario; rather, it can be thought of as a "class" that defines a set of related scenarios, each of which captures a specific course of interactions that can take place between one or more "actors" and the *sud* (an "actor" is an external entity, such as a user, interacting directly with the *sud*).

Notable advantages of this approach include the following:

- use cases are a powerful technique for the elicitation and documentation of the functional requirements of a *sud*;
- as use cases capture the system functional requirements from the user's point of view, they can assist the system designer in both ensuring that the correct system is developed, and validating it;
- use cases can help to master and control the complexity of large projects by decomposing the problem into major functions (i.e., the use cases), and by specifying applications from the users' perspective;
- use cases can provide one with the foundation on which to specify end-to-end requirements for distributed applications.

A number of limitations of this approach have been pointed out in [Meyer 1998, Firesmith 1995]. These limitations arise principally from the observation that use cases are not object oriented. Rather, they allow one to capture functional abstractions that do not necessarily map into objects and classes. However, as pointed out in [Lee 1994], the use case approach is a powerful technique for gathering requirements, and defining problem and system boundaries, effectively; hence, for the purposes of this Report, we have favored this approach.

Thus, in this Report we examine two specific use cases that allow us to capture different scenarios in which the TAPAS platform (i.e., our *sud*) can be used, and to reveal the functionalities the TAPAS platform users can expect from it. The first use case we examine consists of the hosting a *generic* application at an ASP; in contrast, the second use case focuses on the hosting a *specific* application that characterizes the interactions between the TAPAS

platform and its users (namely, an application implementing a fair auction). However, before discussing these two use cases, we introduce below, for the sake of completeness, both the use case abstract model and the terminology we have adopted in the discussion of our two specific use cases.

3.1 Use case abstract model

Each use case includes the above mentioned *actors*, which are external to the *sud*, and may interact with it. Actors can be classified as either *primary* or *secondary* actors, and can represent classes of users, roles that users play, and external systems. The *primary* actor is the one having a goal, and requiring assistance from the *sud* in order to meet that goal; the *secondary* actor is the one from which the *sud* itself needs assistance in order to provide the required services [Cockburn 1997, Fowler *et al.* 1997].

A use case is initiated by a primary actor (e.g., a user), and describes the sequence of interactions, between that primary and the *sud*, which is required in order to deliver the system service that meets the primary's goal. The use case completes successfully when this goal is met.

Possible variants of the sequence of interactions mentioned above, such as alternative sequences that may satisfy the primary's goal, or error handling sequence, are termed *extensions*.

In the use case model, the *sud* is treated as a "black box", and the interactions with it, including the system responses, are as perceived from outside the system [Rosenberg *et al.* 1999].

Note that, in general, a use case is written using a natural language, and the vocabulary typical of the use case domain. Thus, users can easily follow and validate the use cases, and be actively involved in defining the requirements to be met by the *sud* [Malan & Bredemeyer 1999].

3.1.1 Context diagram

The *sud* we are modeling is intended for use from ASPs, who can host, run and maintain customers' applications. The general context within which an ASP carries out its activity can be summarized as follows.

An ASP provides its customers with application hosting facilities on possibly remotely managed servers, and enables the application users to access those applications.

In general, in order to preserve its reputation, an ASP has to provide its customers with sufficient guarantees that a number of QoS requirements (e.g., availability, security, privacy) be effectively met, in the provisioning of its services. Moreover, in order to deliver these services, an ASP may make use of additional services that are offered by partner service providers (e.g. Internet Service Providers, Storage Service Providers); thus, the ASP may require that the services offered by those partners meet QoS requirements.

These requirements are specified within Service Level Agreements (SLAs) established between the ASP and its customers, the ASP and its partners, and, possibly, the customers' applications and their users. (In essence, an SLA is a legally binding contract which defines both the services to be provided, and the metrics that determine the successful delivery of these services.)

In the use cases modeling approach, the relations between an ASP, its customers and its possible partner service providers can be captured and described by means of a so-called *context diagram*; i.e., a diagram showing the system, its actors, and the relevant use cases [Ogush *et al*. 2000].

Figure 9 illustrates the context diagram concerning the generic application hosting scenario, mentioned previously. In particular, this Figure shows the relations between both the ASP and a primary actor (i.e., the hosted *application*, in Figure 9), and the ASP and a secondary actor (i.e., the *partner*, in Figure 9). These relations are governed by the SLAs, depicted in Figure 9 (for the purposes of this discussion we shall assume that no SLA is defined between the application and its end-user).

From the context diagram depicted in Figure 9, it emerges that an ASP requires that the applications it hosts can access the services they need, regardless of the underlying platforms, or organizations, through which these services are provided (i.e., hosted applications do not need to be aware of possible SLAs between that ASP and other service providers).

The *sud* we are concerned with, i.e., the TAPAS middleware platform, can be used to meet this ASP requirement, provided that this platform incorporate a collection of QoS-aware middleware services that meet non functional QoS requirements such as those mentioned above.

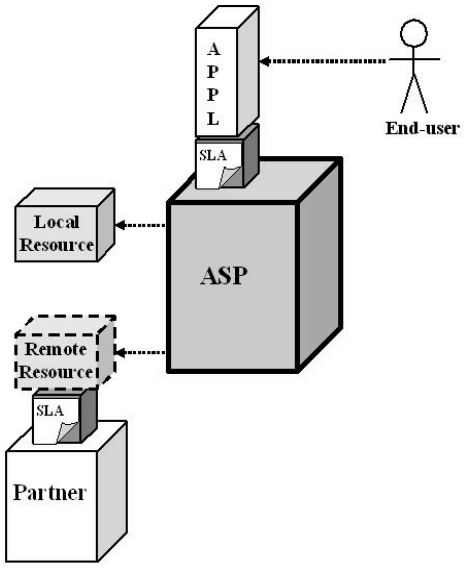

Fig. 9: The Application Hosting Scenario

3.1.2 Terminology

The following terminology is used in the description of the use cases:

- *ASP*: this acronym is used to refer to the TAPAS platform offering access over the Internet to applications and related services, making available both its own resources and those belonging to partner service providers, according to the SLAs.

- *Application*: a program implemented to perform a specific function for use from the end-users or, in some cases, from another application; it requires hardware and software resources in order be executed.
- *Resource*: any addressable unit of information or available facility, offered in a limited supply, used by an application to perform its function. It can be a local resource, such as storage or a processing unit, as well as the network, or a remote service provided by a partner service provider.
- *Customer*: the entity that obtains the service as defined in the SLA; it can be the end-user, the application owner, or the ASP itself, as customer of one of its partners.
- *End-user*: the ASP's customer; i.e., an individual entity or an enterprise using the application hosted by the ASP.
- *Partner*: a service provider having the ASP as a customer, supplying it some services, and providing it with infrastructures (e.g., an ISP that offers network services according to some SLA with the ASP).
- *Service*: what is supplied by the ASP, which hosts the application, or by one of its partners, which provides access to its resources.
- *SLA*: the legally binding contract, which defines the services to be provided and the metrics determining the successful delivery of these services. The SLAs bend the parties, regarding, for instance, security, privacy, responsiveness and measurable quality of the underlying network.

3.2 Use case A: Application Hosting scenario

This use case shows the correlation between the ASP and the end-user, assuming that the goal is to enable the ASP to host the end-user application, according to a contract between them.

The end-user requires the ASP to host an application, using the ASP resources. The ASP allows the end-user to run the application in accordance with the agreements stated in the SLA governing the service provision. This SLA is included in the contract mentioned above.

The ASP monitors the application and the resources it makes available, in order to assess whether the service is being delivered as agreed in the SLA.

If problems arise in the delivery of the service at application run time, it is the responsibility of the ASP to adapt the service provisioning, in order not to violate the SLA. The adaptation can start before the SLA violation point is reached. For instance, it may occur that the SLA is violated for a *short time*, which denotes that the violation point is not yet reached; whereas, if performance remains for too long at a lower level, or it exceeds the SLA defined boundaries, the SLA is considered violated, as illustrated in Figure 10.

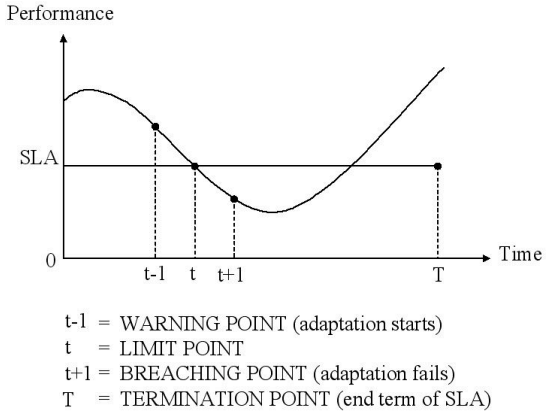

Fig. 10: Performance metrics

Normal termination of the service delivery occurs when this delivery terminates without violations of the SLA. Abnormal termination of the service provisioning may occur when the SLA is violated. (In this latter case, the ASP may be subject to penalties; however, this topic is outside the scope of this Report.)

3.2.1 Main success scenario

1. The end-user contacts the ASP in order to require the hosting of an application.
2. The ASP and the end-user define an SLA (or a set of SLAs) that specify the application hosting requirements.
3. Both parties agree on, and sign, a contract containing the SLA(s). This contract defines the terms and conditions under which the application hosting service is to be supplied.
4. The ASP runs the application, according to the SLA, and using the available resources.
5. The ASP may accesses remote resources provided by its partners.
6. The end-user uses the application hosted by the ASP.
7. The ASP monitors and records the performance of the application to assess whether or not the SLA is being honored.
8. The contract terminates normally when it reaches the closing terms.

3.2.2 Extensions

5.a. An ASP partner violates the SLA it has with the ASP, causing difficulties to the ASP in the service delivery to its customers.

5.a.1. The ASP complains with the partner.

5.a.2. The ASP adapts its services.

7.a. The ASP violates the SLA with the end-user for a short time:

7.a.1. ASP adapts its services.

7.b. The ASP violates SLA for a long time:

7.b.1. the end-user complains with the ASP

7.b.2. the contract is breached abnormally. End use case.

3.3 Use case B: Auction Hosting scenario

In this use case, the ASP is hosting an electronic auction application. It interacts with the following primary actors (i.e., its customers): end-users playing roles of auctioneer, sellers, and bidders. At the same time, it interacts with partner services providers, which are the secondary actors. The goal is to make the system host the auctioneer's application, so that sellers and bidders can participate to the auction in order to exchange goods. The context diagram illustrating this scenario is depicted in Figure 11.

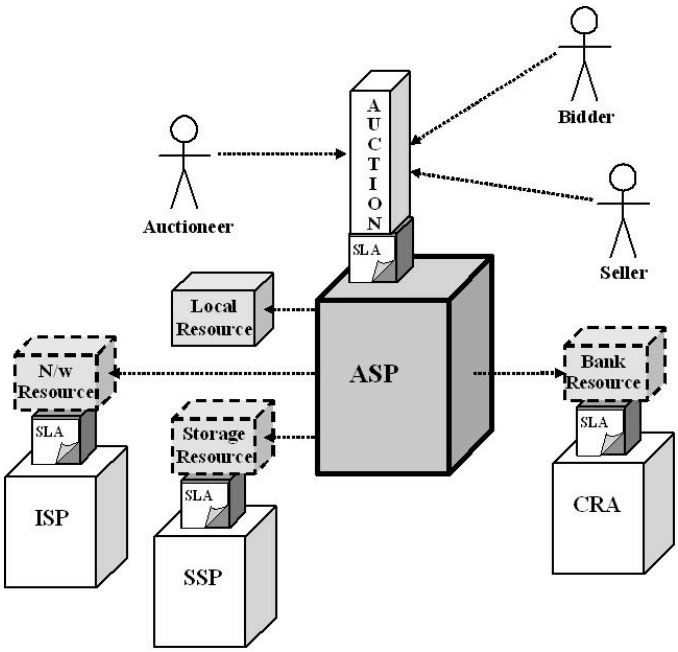

Fig. 11: The Auction Hosting scenario

End-users contact the ASP with different purposes. Namely, the auctioneer will contact the ASP in order to have his auction application hosted by the ASP (in accordance with some agreed contract); the seller(s) in order to sell goods through the auction application, and the bidders in order to bid for those goods.

The bidders bid for the items on sale, and the ASP starts the auction collecting all the necessary data. The auction is terminated if there are no more bids for the item within a timeframe from the last valid bid.

In hosting the fair auction, the ASP contributes to preserve the fundamental *fairness* property of an auction, specifically: "all participant bidders in the auction must have an equally fair chance for submitting a successful bid, and that all participant sellers must have an equally fair chance for selling their items" [Ezhilchelvan *et al.* 2001].

In his job the ASP is helped by its partners, which it must coordinate and collaborate with for the success of the electronic auction. There are different partners, one for each required service for the auction's establishment. Specifically there are service providers such as the ISP, offering communication resources, the Storage Service Provider (SSP), offering storage resources, and the Credit Rating Agency (CRA) that keeps in contact with retails banks for the purchase of the item. The ASP sends requests for proposals (RFP, a sealed proposal used for purchases of items or services) to such a provider's component, asking the CRA for financial services.

In this way there is a chain of calls between subsequent service providers. Regarding the service providers, there is a certain level of standardization offering real benefits. Service providers can easily be exchanged if switching from one service provider to another does not lead to further interface programming [Beckman *et al*. 2002]. At the closing terms, if the contract with the auctioneer is honored, the service delivery ends normally; otherwise, it can be terminated abnormally.

3.3.1 Main success scenario

1. The end-user, playing the role of the auctioneer, requires the ASP to host his auction application, according to a commonly agreed SLA.
2. To be on-line, the ASP uses the network services offered by the ISP
3. The ASP starts hosting the auction application
4. An end-user calls in the ASP to use the auction application with a request for selling an item, registering himself as seller.
5. The ASP records the seller's data (e.g., name, address, details of the item on sale).
6. End-users call in to place bids, registering themselves as bidders.
7. The ASP records bidders' data (e.g., name, address, requested item on sale).
8. The ASP provides the bidders with information on selected items (e.g., price, delivery date).
9. Bidders participate to the auction run by the ASP by submitting bids.
10. The winning bidder signs for order.
11. The ASP calls in the CRA, requesting for approval of the financial transaction required in order to sell the item.
12. The contract between the ASP and the auctioneer for hosting the auction application ends successfully when it reaches the closing terms.

3.3.2 Extensions

1.a. During the duration of the contract, the SLA is violated by the ASP for a short time:

1.a.1. the ASP adapts itself in order to honor the SLA.

1.b. SLA is violated by the ASP for a long time:

1.b.1. End-users complain with the ASP

1.b.2. Contract is abnormally breached. End use case.

2.a. Network services are not efficient and ASP may have difficulties in offering the requested service to its customer:

2.a.1. the ASP complains with the ISP;

2.a.2. the ASP adapts its service according to the modified conditions.

5.a. There is not enough local storage space:

5.a.1. The ASP uses the service offered by the SSP, its partner.

9.a. The ASP does not preserve fairness in delivering the bids:

9.a.1. bidders complain with the auctioneer;

9.a.2. the auctioneer complains with the ASP

9.a.3. Contract is abnormally breached. End use case.

11.a. CRA does not give the approval:

11.a.1. Item is not sold.

3.4 Requirements analysis

In general, the requirements analysis indicates how a *sud* should support the end-users' activities, provides a design strategy for building that *sud* [Robertson 1999], and lead to a well founded architectural model [Verma 2000]. .

For the purposes of this Report, the requirements analysis of the two use cases we have described discloses the two sets of user requirements concerning the necessary system functionalities to be developed (see Subsections 3.4.1 and 3.4.2, below), and two sets of requirements that derive from the activities of our industrial project partners [Beckman & Oleneva 2002] (see Subsections 3.4.3 and 3.4.4, below).

3.4.1 Requirements from use case A

Use case A reveals the following requirements:

R.1.1 The system must be able to interpret the SLA specification included in the contract agreed by the parties. The SLAs' specification has to be done according to a modeling language which is sufficiently expressive to include all aspects (i.e. both technical and business related aspects) of the service provisioning.

R.1.2 The system must run the application using both its own resources and those hired from its partners, enforcing the agreed parameters included in SLA.

R.1.3 The system must monitor the hosted application and the service delivered to the end-user, and must adapt itself to changes that may occur during the service provisioning, always honoring the agreed contract and SLAs.

R.1.4 The system must monitor the services delivered by its partners, in order to honor the SLA with its own customers.

R.1.5 The system must periodically record the performance achieved.

3.4.2 Requirements from use case B

The use case B highlights the following requirements:

R.1.6 The system must collect data reliably, providing non-refutable statistics and trustworthy reports.

R.1.7 The system must use a secure channel to contact the CRA in order to provide safe transaction and management of private information.

R.1.8 The system must undertake the role of coordinator, integrating its services with the ones offered by ISP, CRA, and SSP. Moreover it must provide a good level of standardization of procedures.

R.1.9 The system must eliminate lack of trust in e-business services, using mechanisms of a Certification Service Provider (CSP) and services of Trusted Third Parties (TTP).

3.4.3 Technology requirements

From a technology perspective, the requirements originate from the objective to extend current industry standards and to expand different aspects of the service provisioning.

R.2.1 From the modeling languages perspective, especially for the purposes of providing a standard model for SLA, it should support the use of XML.

R.2.2 At the middleware level, the software industry currently focuses on component based development. In this context, a widely used framework is the J2EE technology. However, there may well be scope for investigating the uses of alternative technologies, such as the CORBA Component Model.

R.2.3 At the network level, the system can make use of reliable multicast communication paradigm, in order to support application requiring many to many communication.

3.4.4 Market requirements

The last set of requirements placed on the TAPAS platform originates from the need for successful marketing. These requirements derive from the economic demands of the three major stakeholders of the market domain to which the project is targeted: namely, ASPs, ASP customers, and application developers.

R.3.1 The system must be useful to the ASP consortium. This entails that the system must reduce operation and maintenance costs by automating such procedures as negotiation, implementation, and monitoring of the SLAs, as well as generating data for statistical analysis.

R.3.2 The system must protect the customers' business, offering assistance to the setting up of the business, in consulting and software development. This entails that the system should be able to support all phases of the software development process, enhancing open source containers, building customers' applications, and deploying and running them.

R.3.3 The system must provide facilities for easy development of third party business, which implies it should support business-to-business integration, bringing together companies that can collaborate in order to satisfy the market's demands.

3.5 Mapping requirements to services

The requirements captured by the previous analysis can be mapped to the services of the TAPAS middleware platform that meet them. Each group of requirements denotes distinct system services that can be identified by means of their interface description, as discussed below.

3.5.1 Interpreter Service

Requirements **R.1.1** and **R.2.1** can be met by an *Interpreter Service* that converts the SLA high level specification into machine-readable format.

The Interpreter Service can use a modeling language that can express unambiguously the properties of the IT services declared in the SLAs, and included in the contractual agreement between the parties. A good example of this modeling language is SLAng [Lamanna *et al.* 2002], an XML language for capturing Service Level Agreements. This language provides means to describe technical and non-technical characteristics of a service, including a so-called Service Level Specifications (SLSs) that defines the QoS properties a service is to possess and the related set of metrics with which the service provisioning can be measured.

The *Interpreter* isolates the SLSs within the SLA, and generates specific objects (e.g., *policy objects* [Kakadia 2001], *vector of performance* [Menasce` 2001], a configuration file including <parameter-value> pairs) which can be used by the Configuration Service, described in the next Section, for setting up the environment in which the application can be executed.

Moreover, the *Interpreter* can make available the SLS to specific platform services that monitor the adherence of the service delivery to that SLA.

3.5.2 Configuration Service

Requirements **R.1.2, R.2.2** and **R.3.2** can be met by the *Configuration Service*, responsible for the management of the resources necessary to support the execution of the distributed application, based on the QoS parameters specified in the SLS. Its principal responsibilities include the following three services:

- *Resource Discovery*
- *QoS Negotiation*
- *Resource Reservation*

These three services are concerned with the configuration of the system and, in particular, the configuration of the resources required to achieve certain QoS levels. The discovery service is responsible for discovering the available resources, and assessing whether these resources are sufficient in order to meet the QoS application requirements.

Given a certain QoS required by the application, and a certain QoS which can be offered by the middleware platform, a mechanism is required that negotiates the QoS levels and resource allocations with the underlying system, and, possibly, other applications. This mechanism is is implemented by the Negotiation Service. This Service will generate an agreed QoS; i.e., a sort of contract which is established between the application and the middleware. This QoS contract can be thought of as the QuO contract [Zinky *et al*. 1997, Loyall *et al*. 1998], previously described, which QuO's delegates process in order to make a QoS-aware remote method invocation.

Given this contract, the last operation the Configuration Service does is to reserve the resources identified and negotiated in the previous phases.

3.5.3 Reporting Service

Requirements **R.1.5, R.1.6** and **R.3.1** can be met by the *Reporting Service*. It periodically collects samples of performance, specifically the metric values expressing the technical measures of the services provided by the system itself or by its partners. The collection of data is made through a set of distributed *Sensors*, such as those used in ControlWare [Zhang *et al*. 2002], that periodically measure the achieved performance values.

Once the values are measured, the *Reporting* gathers and records them in order to be used for statistical analysis, as well as to provide trustworthy reports for the customers and the partners, and furthermore to represent an irrefutable proof in case of disagreements between parties. The implementation of this service must involve dedicated mechanisms to providing trust issues.

3.5.4 Controller Service

Requirements **R.1.3** and **R.1.4** can be met by the *Controller Service*. This is the service in charge of monitoring and adapting the hosted application and the middleware services used to run it.

Adaptation might be caused by changes in the require output quality of service levels, in the input quality of service levels, in external demands for resources (due to new request for service or change in resource requirements), in system-wide resource reallocation. Hence, it is necessary to monitor, at run time, the quality of service that can be achieved and, eventually, to adapt these QoS levels in accordance with the changes occurred. The Controller Service’s responsibilities are the following:

- *QoS Monitoring*
- *QoS Adaptation*

The QoS monitoring activity is carried out in collaboration with the *Reporting* which gives data collected by the sensors. These data are compared with the QoS levels specified by the *Configuration*. For example, it checks if the output quality achieved falls below the required level. At this point, whenever a warning condition is detected, the *Controller* must enforce the required service, adapting the run-time environment and the resources, without affecting any other application running on the middleware platform, and considering that the performance are mainly composed by a set of concurrent factors, for instance, a message Delay is due to the network Round Trip Time (RTT), to the server or client Delay, to the database I/O Performance Delay or to the external memory Access Time.

In case the adaptation is necessary the *Controller* must decide which strategy to apply. A theoretical basis for the design and implementation of this adaptation-based service can be the *Feedback Control Theory*, which recent results indicate to be an effective instrument for the monitoring and adaptation of software performance in highly unpredictable environments [Ferrari G. 2002].

3.5.5 Coordinator Service

Requirements **R.1.8**, **R.2.3** and **R.3.3** can be met by the *Coordinator Service*. With this service the ASP addresses the needs of cooperation between different enterprises and, more specifically here, enterprises providing IT services, as the ASP itself, ISP, SSP and general customers, coordinating a chain of calls between subsequent service providers and end-users. The ASP fulfils this function detecting whether parties are observing the agreements for the service provisioning; it receives the periodical reports from the *Reporting* and compares them with the agreements written in the contracts previously stipulated.

It must be remarked the two different kind of business in which the ASP is involved. On one hand, regarding the contracts it has with its partners, the ASP appears as customer, on the other, regarding the contracts with the end-users, ASP acts as provider. So that, in the former case, it must monitor the received services, in order to verify that its rights are respected, but above all, in order to fulfill the obligations with the end-users, looking after the agreements with the customers and trying to adapt to any change of conditions. Whether any violation is detected it must be registered and notified to the interested parties, using an Event Notification System efficient enough to reveal all the problems arising in the service provisioning. The interested parties are the ones involved in the contract, end-users or partners; furthermore in our future scopes we will include the *x-contract*, defined in the deliverable D5 of the TAPAS project.

3.5.6 Security Service

Requirements **R.1.7** and **R.1.9** can be met by the *Security Service*, which is responsible for secure procedures, essential for assuring safe Business-to-Business transactions and management of financial data. It uses registration and authentication mechanisms to undertake the information exchanged with the CRA. As well as the *Reporting,* it must eliminate lacks of

trust, making use, among all, of trustworthy operations in the collection and management of private data, for instance *B2Bobjects* [Cook *et al.* 2002].

3.6 High level System Architecture

Figure 12 illustrates how the various services interact within the TAPAS platform in order to meet the elicited requirements.

The approach used, starting from use cases, which drive the requirements analysis, and the following mapping of requirements to middleware services, provides solid guarantees on the correctness, motivating the designing resolutions and validating the presented architecture.

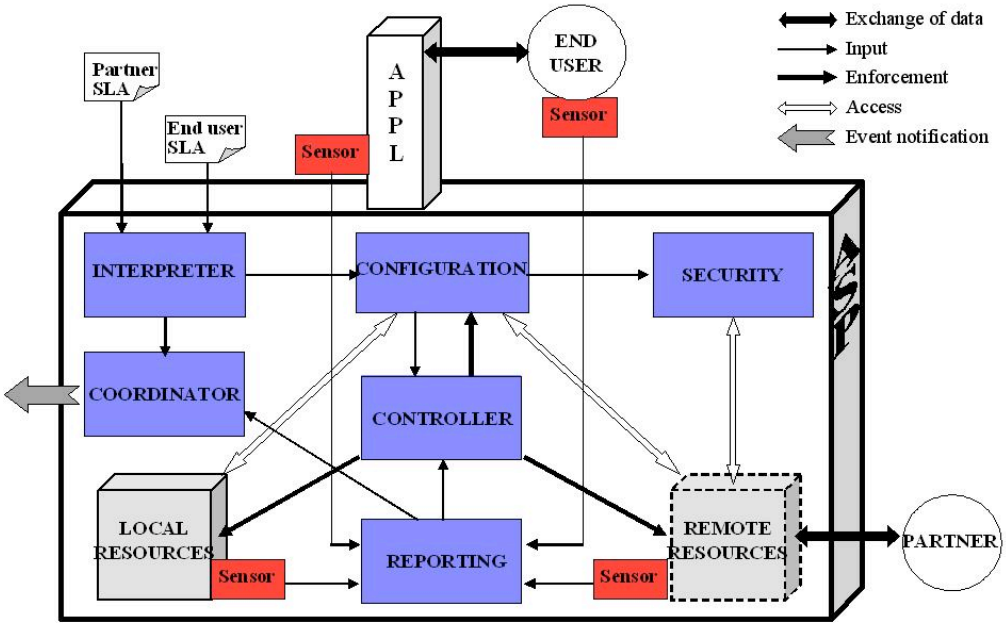

Fig. 12: Inside the "black box", the middleware services

4 QoS-aware Application Server Architecture

In this Section we examine how to instantiate the architecture, described in the previous Section, using the Java 2 Enterprise Edition (J2EE) component based technology.

4.1 System Model

As previously stated, ASPs are organizations that offer access to applications, such as software, and related services on a rental basis via Internet. They supply a complete infrastructure for their customers and they manage the inter-network and all the applications which clients wish to run. In order to carry out these duties, the ASP may use a cluster of different machines running heterogeneous operating systems. The machines can be located in different places, arbitrary faraway from each other, so that they compose a wide area network infrastructure.

A possible system model is shown in Figure 13, where an ASP hosts distinct applications such as electronic auctions, video conferences, and games over Internet. It might be possible that an application uses functionalities or components of one or other applications hosted by the ASP

itself. This can be done through the use of the middleware platform above designed, which provides services so that distributed application interactions are carried out in a transparent, secure, trusted and QoS-aware manner.

The platform consists of a group of application servers that exchange information with each other inside the group itself. The application servers are execution environments providing a number of core services capable of meeting specific non-functional requirements of the applications. Each application server may be hosted inside a single machine; one machine can run more than one application server (this can be useful for emulating a cluster of machines).

As the current trend is to design applications constructed out of a set of components, it may be possible to distribute the instances of these components in different application servers. Thus, Web components (e.g. servlets, Java Server Pages) can run on a Web Server, hosted inside an application server, and the business logic components can be maintained separate and run on another application server. Figure 13 shows the example of the electronic auction, which is a particular demanding application in terms of QoS.

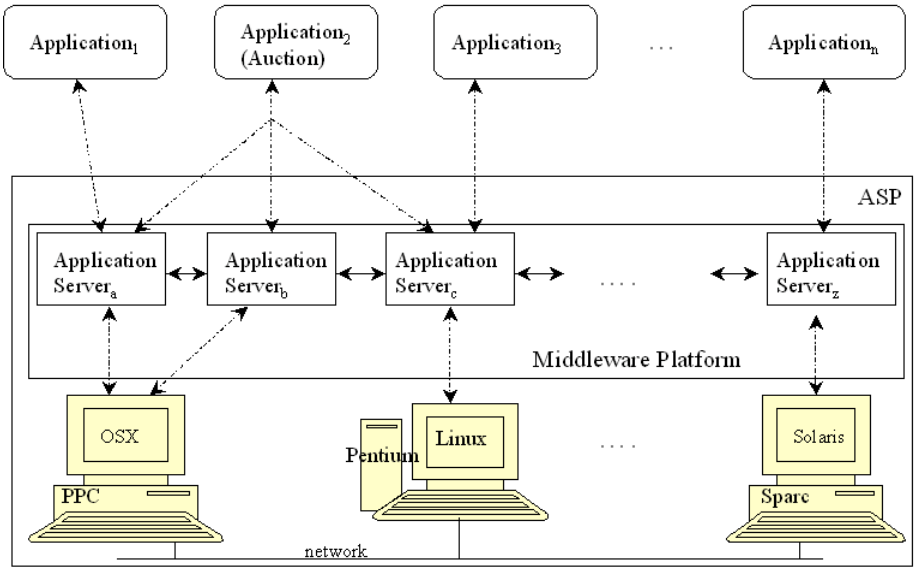

Fig. 13: System Model

4.1.1 Auction Scenario

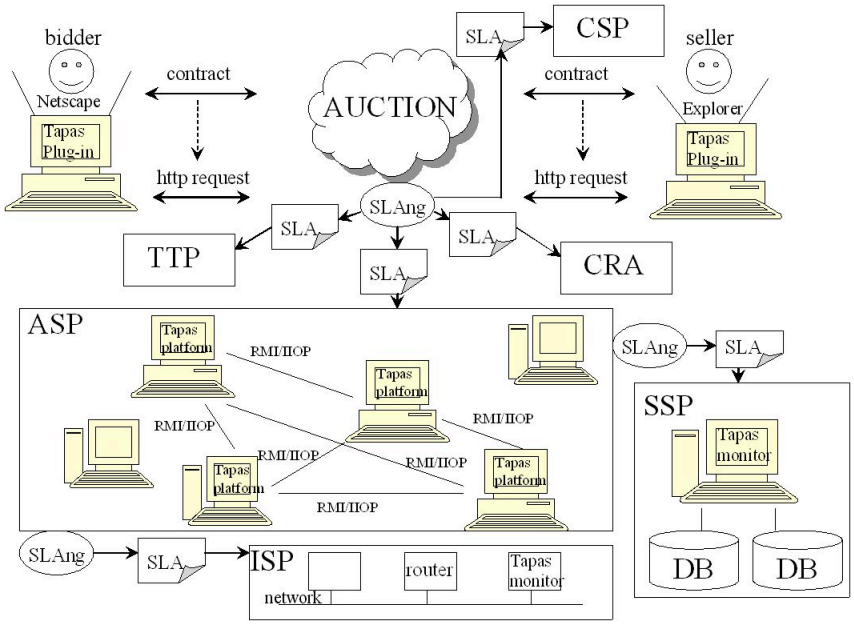

Fig. 14: Auction Scenario

Figure 14 illustrates the auction scenario that has been abstractly described in the Section 3.3, where different entities interact with each other in order to complete auction transactions.

To manage the interactions between bidders and sellers, the application owner makes available the electronic auction application. In this way, bidders and sellers register themselves with the auction owner, by signing a contract. This contract entitles the end-users to use the auction system: they can access the information, generally, through HTTP requests and they can use a software plug-in (*Tapas plug-in* in Figure 14) which basically monitors the clients' activities.

The application owner might have his/her own infrastructure to host the auction application, or might ask an ASP for providing him/her with a suitable platform that can execute that auction application. In the latter case, an SLA is established between the two organizations to which the ASP and the application owner belong; this SLA can be described using *SLAng* [Lamanna *et al.* 2003].

At this point, the ASP might have all the needed resources to execute the auction, or may require network or storage resources, for example from an ISP and the SSP, respectively. Again, this requirement can trigger the stipulation of SLAs whereby every term for the provisioning of the service is specified.

The ASP may use a subset of its machines for running the middleware platform, which we have called *TAPAS platform*, with the standard and new QoS-aware services provided. These services communicate with each other using a protocol such as RMI/IIOP.

In order to implement the middleware platform, in next Section we will describe a study about component based technologies.

4.2 A proposed architecture implementation

As already mentioned, state of the art application services can be developed using components based technologies, such as those provided by J2EE, Microsoft .NET or the Object Management Group's CORBA Component Model.

These component oriented middleware promote the use of *containers* to host component instances. A container is a runtime environment which provides the system-level services to the components it contains. Thus, it is responsible for using the underlying middleware services for communication, persistence, transactions, database management, security and so forth.

Our goal is to implement the middleware architecture of our model using just these kind of concepts and services. Before explaining the new services the middleware requires, in order to meet application specific QoS requirements, let us describe the component oriented technologies we are planning to use for the implementation of our proposed architecture.

4.2.1 Java 2 Enterprise Edition

As shown in Figure 15, the J2EE platform uses a multi-tiered distributed application model. This means that the application logic is divided into components according to function, and the various application components that made up the J2EE application, are installed in different machines depending on which tier, in the multi-tiered J2EE environment, the application component belongs [Bodoff *et al*. 2001, Shannon 2001].

The J2EE platform consists of the following three principal tiers:

- *Client Tier* — This tier includes the client components running on the client machine.
- *Middleware Tier* — This tier is structured in two further tiers; namely, the *Web Tier* and the *Business Tier*. The former includes the Web components, the latter the Enterprise Java Beans (EJB), both running on the J2EE server.
- *Enterprise Information System (EIS) Tier* — The EIS software runs on the EIS server

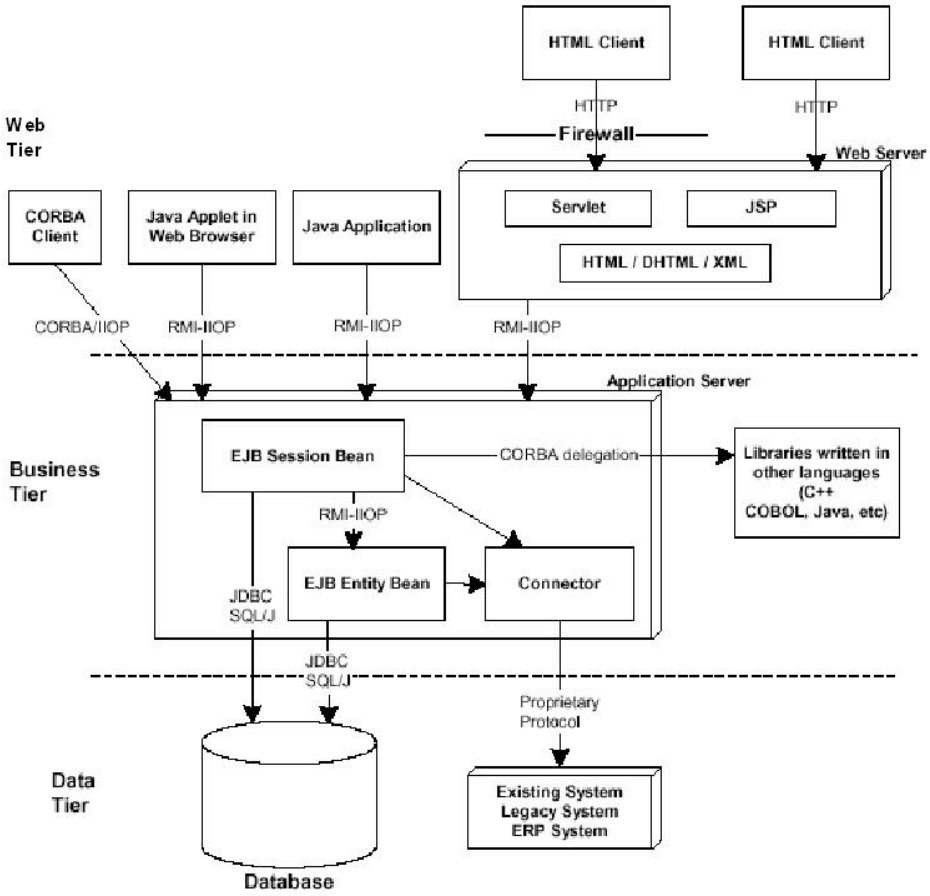

Fig. 15: J2EE platform

4.2.1.1 Components and Containers

Starting from the client tier, the client components include:

- *Application clients* — They are hosted inside an *application container* which is, in turn, responsible for managing the execution of the application client components
- *Applets* – They are hosted inside an *applet container* which, in turn, manages the execution of applets and consists of a Web browser and Java Plug-in together, running on the client

J2EE Web components can be either *servlets* or *JSP pages*. Servlets are java classes that dynamically process requests and produce responses. JSP pages are text-based documents that execute as servlets. These Web components are hosted inside *Web containers* that take care of the execution of servlets and JSPs.

Finally, business components are basically constituted by Enterprise Java Beans (EJBs) hosted inside an *EJB container*. There are three types of EJBs:

- *Session Beans* — Represent a transient conversation with a client
- *Entity Beans* – Represent a persistent data storage in a row of a database table

- *Message Driven Beans*– Combine features of a session bean and a Java Message Service (JMS) listener, allowing a business component to get messages asynchronously

This J2EE architecture has been implemented by different working groups producing application servers free to download. In this report, we will focus on two of them: JBoss and JOnAS.

JBoss is an open source application server. It integrates a set of services for a full J2EE-based implementation. Starting from the Java Management eXtensions (JMX), the JBoss group built a microkernel based server [JBOSS 2003].

JOnAS is also a Java open source application server conforming to the J2EE specifications [JONAS 2003]. It is highly modular and can be used:

- *As a J2EE server*– For deploying and running EAR (Enterprise ARchive) applications
- *As an EJB container*– For deploying and running EJB components
- *As a Web container*– For deploying and running JSPs and servlets

Next Section focuses on QoS-aware middleware functionalities that we believe are necessary in order to meet QoS requirements specified within SLAs.

4.3 QoS-Aware Application Servers

Both the J2EE application server and its relative open source implementations are designed in terms of services. A service, typically, provides system resources to containers. As stated before, these services include persistence communication, database managements, transaction, security and so forth.

Moreover the TAPAS platform must provide QoS-aware execution of components so that application servers themselves rely on the QoS-aware middleware services introduced in the Section 3.5. To achieve our target, J2EE platform must be extended in order to enable the monitoring and the enforcement of QoS.

4.3.1 How to include QoS-Aware services in J2EE

The J2EE platform introduces an important architectural concept, namely the container; i.e., an environment hosting application components instances and providing a set of middleware services. In fact every container uses the underlying middleware services (e.g. Java Message Service (JMS), Java Transaction Service (JTS), Java Authentication Authorization Service (JAAS), Java Naming and Directory Interface (JNDI), etc...) that, in turn, rely on Java 2 Standard Edition (J2SE), as it is shown in Figure 16.

The J2EE containers provide the Application Programming Interfaces (APIs) that define the contract between J2EE application components and the J2EE platform. However, there are other important contracts that are established between the J2EE platform and the service providers that may be plugged into a J2EE product.

These other contracts are managed by the *Connector* (shown in Figure 15), a J2EE standard for integrating J2EE products and applications with heterogeneous Enterprise Information Systems (EISs). Specifically, the Connector enables the existing EIS of a vendor to provide a standard resource adaptor for its information system. The Connector architecture defines the services that the J2EE-compliant application must provide. These services, namely the *transaction manager*, *security* and *connection pooling*, are delineated by three Connector system-level contracts. The first one, concerning the transaction manager, provides one with the ability to manage transactions across multiple EIS resource managers; the second one, concerning the security, enables secure access to an EIS and protects the resources managed by that EIS; finally, the third one, concerning the connection pooling, enables an application server to pool connections to an underlying EIS, which may be crucial in order to create a scalable application environment. These three system contracts together can be considered as a Service Provider Interface (SPI).

Hence, our overall objective is to add new services in order to provide QoS-aware containers. First of all, here we present the Configuration Service (CS) and the Controller Service (CTRL), that have their own SPIs used to interact with the platform and the new APIs used, at the upper level, by application components, in order to satisfy particular QoS requirements. Moreover, these new services must be added to the entire platform so that containers use them in the same manner they do for the other standard services. Figure 16 highlights the integration of the new QoS-aware service added in our design, with the tiers of the overall J2EE architecture (illustrated in Figure 15).

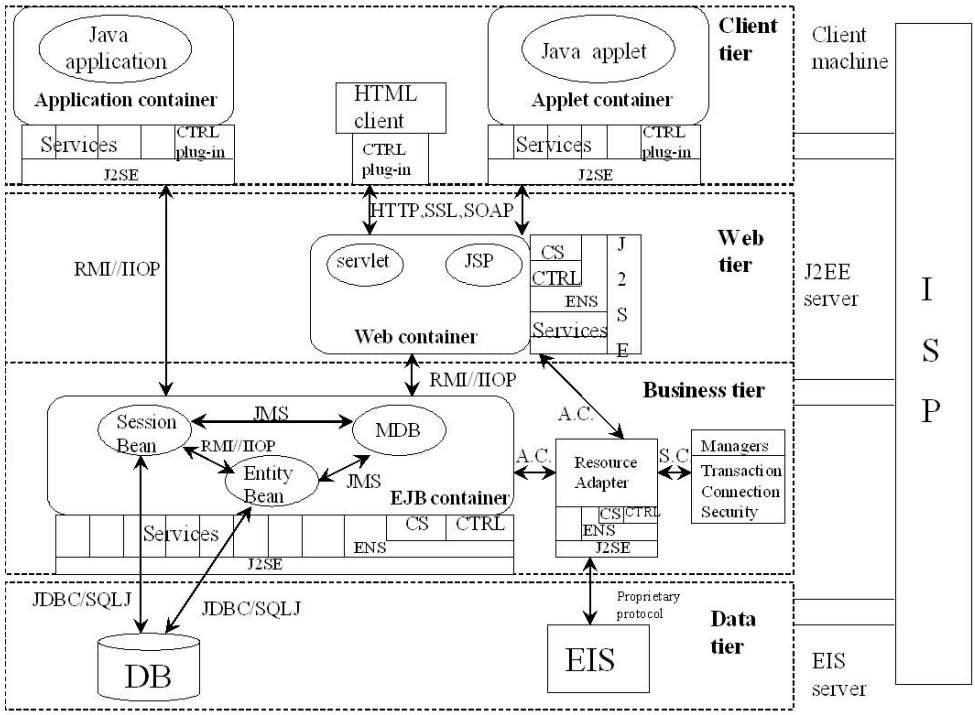

Fig. 16: J2EE: QoS-aware containers

4.3.2 How to extend JBoss

The JBoss' goal is to provide the full Open Source J2EE stack. To achieve this objective, it uses a software called JMX (Java Management eXtension) which is a powerful tool used for software integration. JMX provides a software bus that allows one to integrate modules, containers and plug-ins. Figure 17 illustrates how JMX is used as bus through which the components of the JBoss architecture interact [Stark & Jboss Group 2002].

The basic idea is to produce some independent QoS-aware modules (with their own APIs and SPIs) such as Configuration Service Module (CS in Figure 17) and Controller Service Module (CTRL in the Figure), and use the JMX for integrating them with the other standard services already provided by the platform.

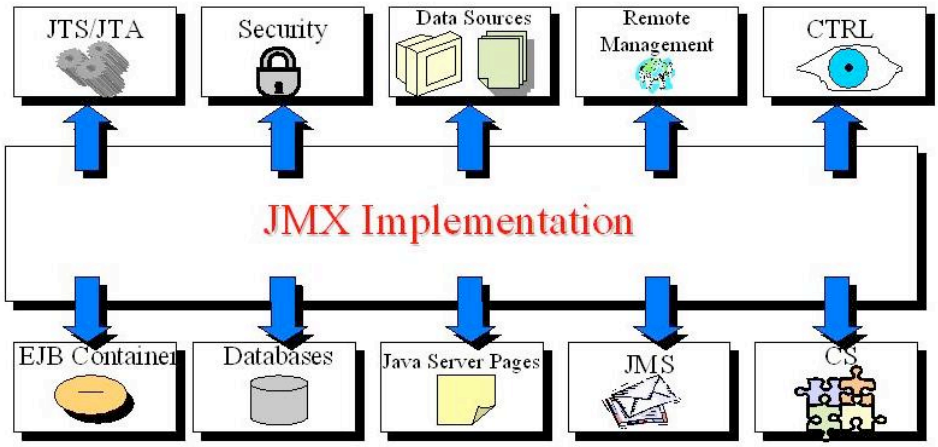

Fig. 17: JBoss JMX Microkernel

4.3.3 How to extend JOnAS

JOnAS is a pure Java open source application server and includes some advanced and important features to the implementation of all J2EE related standard. These characteristics are the following [Checcet & Marguerite 2002]:

- *Management* – JOnAS server management uses JMX (as JBoss does) and provides a servlet based management console named Jadmin (Figure 18)
- *Sevice* — It allows to apply a component model approach at the middleware level and makes easy the integration of new modules. It also allows to start only the services needed by a particular application, thus saving useless system resources
- *Scalability* — JOnAS integrates several optimization mechanisms for increasing the server scalability
- *Distribution* — JOnAS is working with two distributed processing environments, RMI (Remote Method Invocation) or Jeremie, the RMI personality of an Object Request Broker called Jonathan

The main principle for defining a new JOnAS service is to encapsulate it in a class whose interface is well known by JOnAS. More precisely, such a class allows to initialize, start and stop the service. Then, in order to make JOnAS aware of such a new service, some properties files should be changed accordingly. So, again, the basic idea is to produce separate modules, with their own APIs and SPIs and, in this case, make them available as a class (Figure 18).

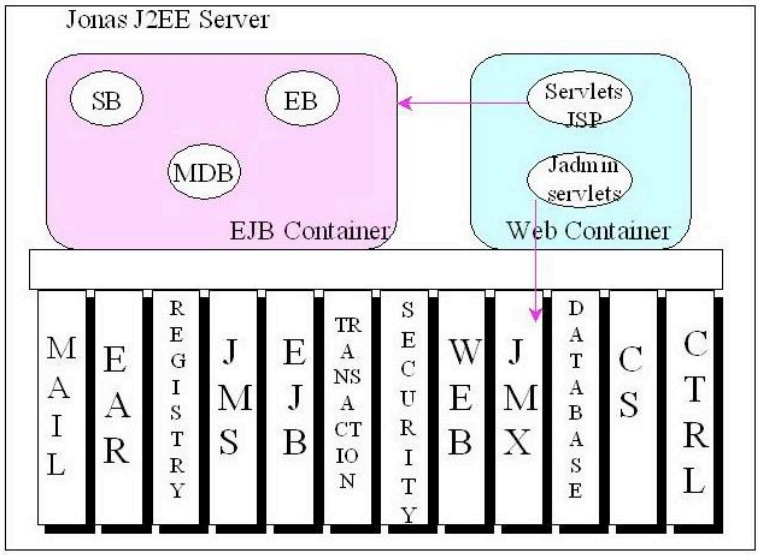

Fig. 18: JOnAS Architecture

4.3.4 Packaging

Three important phases can be identified for producing the final component application that is executed in any of the application servers described earlier. These three phases are (Figure 19):

- *Coding Phase*
- *Deployment Phase*
- *Running Phase*

The first is generally carried out by an Application Developer who is responsible for producing all the final application source files compiled, together with the XML document where the trust and QoS requirements, previously derived by the SLA through the Interpreter Service, are included.

These files are then made available to the Deployer who is responsible for carrying out the deployment phase.

The deployment needs before that all the J2EE-compliant application components be packaged and bundled into a J2EE final application. This application, together with its modules, is then delivered in an Enterprise Archive (EAR) file, which contains deployment descriptors that are XML documents describing the components' (i.e., EJB components, Web components) deployment settings. These settings specify the application component's external resource requirements, security requirements, environment parameters and so forth. Then, the

deployment descriptors must be parsed and the environment configured. To this end, our CS can be enabled, kicking off the protocol for discovering, negotiating and reserving the resources required by the application.

Once the platform to be used is configured, the same Deployer starts up the execution of the newly installed and configured application. The application has to be monitored, at run time, to guarantee that its execution is respecting the SLA. The monitoring is a responsibility of our CTRL that will apply appropriate adaptation strategies if changes in the environment are detected.

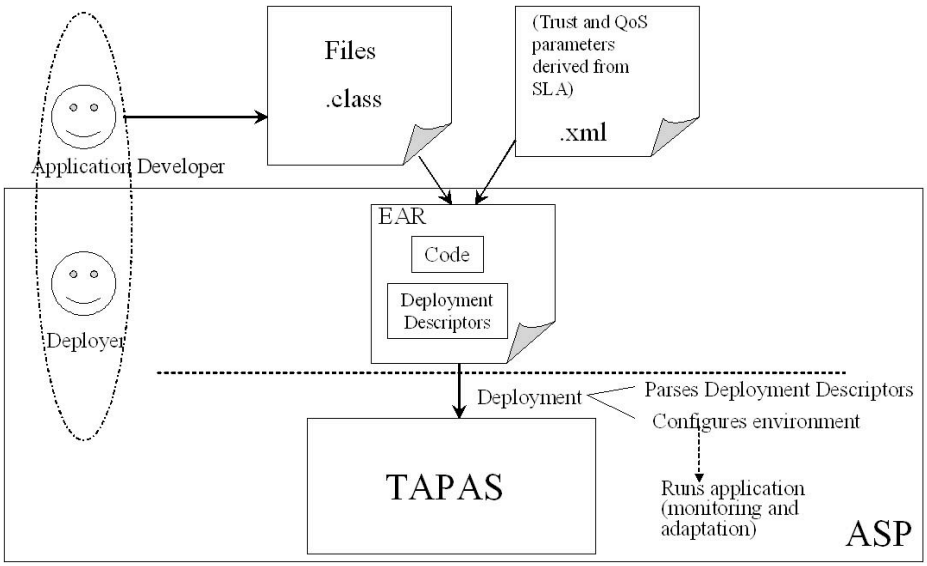

Fig. 19: Coding/Deploying/Running

4.4 Configuration Service implementation

The Configuration Service (CS) is one of the services we have introduced in our middleware architecture in order to make it QoS-aware. Its principal responsibility is to configure the platform used to support the execution of distributed applications. To this end, the Configuration Service performs the following three principal activities: resource *discovery*, resource *negotiation* and, finally, resource *reservation*.

The resource discovery activity allows the CS to locate the resources required in order to execute the application. As the resource discovery completes, the negotiation activity assesses the availability of these resources. Finally, as the resource availability has been assessed, the reservation of the resources identified as available is carried out (i.e., these resources are booked in order to support the application execution).

In order to assess the availability of the resources, each application server in our architecture implements a Reporting Service (ES), responsible for monitoring and reporting the state of the resources, local to its home application server. Specifically, the RS, through the use of sensors such as those described in [Bergmans *et al.* 2002], maintains reports on the state of the local resources, in persistent storage. These reports can be used by the CS for assessing the resource availability.

An example of CS implementation is suggested by the protocol described in Figure 20, using a pseudo-code. The CS is started at application server start up time. At that point, it is ready to get application requests. When the SLS specifications are available, at deployment time, the CS is enabled. This leads to the execution of the CS protocol (i.e., the above mentioned discovery, negotiation and reservation of the required resources).

In our example, we assume that each application server incorporates an instance of the CS, which resides at some well-known address within its home application server, and supports the construction of the application run-time environment in that specific server. If a group of application servers is required to work together to host an "extended application" (i.e., an application running on different servers), the CSs in this group of servers can coordinate, using group communication technology, in order to set up the required application run time environment across the various application servers in the group.

Within this scenario, the enabled CS, with the SLS as input, can declare itself as the group *coordinator*, and send a request for resource availability to the other CSs in the group, providing them with its own input SLS. (The SLS is transmitted for survivability purposes; i.e., if the coordinator crashes, an alternative coordinator can be elected to complete the required configuration, based on the SLS it possess.) At this point every CS in the group, including the coordinator, enquires for the locally available resources. This brings every CS to book its own available resources, and to send its availability back to the coordinator. Then, the negotiation can be started: the coordinator collects the information received in order to derive an *agreed QoS* that is, as introduced earlier, a contract between the application and the middleware platform. For instance, this contract may include which servers to use, and how much resource to spend, for each type of resources involved in the process. These values can be derived from appropriate calculations that take into account both the application requirements specified within the SLS, and the states of the resources obtained by the RS.

Once producing this agreed QoS, the coordinator is ready to confirm the reservation of the resources previously booked. Finally, the containers are instantiated inside the application servers so that they can be used to host application components.

```
START UP TIME
    START_UP()
    begin
       CS_id = start_CS();                          /* starts the Configuration Service */
    end.

DEPLOYMENT TIME
    ENABLE_CS(SLS(Application))
    begin
       I am the coordinator                        /* declares itself as coordinator of the group */
       membership = getMembership(coordinator_id);  /* gets the membership of the group giving its id */
       /*DISCOVERY PHASE: returns the availability of each CS*/
       available_resources = discovery(SLS(Application), membership);
       /*NEGOTIATION PHASE: produces agreed QoS with SLS & available resources*/
       agreed_QoS=negotiation(SLS(Application), available_resources);
       /*RESERVATION PHASE: confirms the reservation */
       reservation(membership);
       return agreed_QoS;
    end.

    DISCOVERY(SLS(Application), membership)
    begin
      /* the coordinator gets its local resource state */
      /* the coordinator books its available resources */
      /* for each member of the group asks for their availability giving the SLS */
         /* each member stores the SLS, gets its own resource state, books its available */
        /* resources and returns its availability */
      /* returns the available resources */
    end.

    NEGOTIATION(SLS(Application), available_resources)
    begin
      /* compares SLS(A) with available_resources    */
      /* builds the agreed QoS */
      /* return agreed QoS (the platform to be used)*/
    end.

    RESERVATION(membership)
    begin
      /* for each member of the group */
      /* confirms the previous booking (YES, NO, NOT COMPLETELTY (how much)) */
    end.
```

Fig. 20: Configuration Service protocol

Different types of resources must be orchestrated by the CS. One of them is the database that can be local or remotely distributed. Next Section describes some replication techniques for high availability regarding databases and application servers.

4.4.1 Component replication: a case study using EJBs

One of the responsibilities of a service provider is to guarantee the *availability of the service provision*, i.e. the time during which a service or an application is guaranteed to be available. It can be defined by a given period of time or a percentage or a combination of both [Beckman *et al*. 2002].

In order to satisfy this QoS requirement, the middleware platform is capable of applying techniques of component replication. Specifically [Morgan *et al*. 2002] present a study whereby they examine how replication for availability can be supported by containers so that components that are transparently using persistence and transaction can also be made highly available (see attached paper for details). They take the specific case of EJB components for managing replication and they consider *strict consistency* (i.e. to require that the states of all available copies of replicas are kept mutually consistent).

The aim here is to investigate how existing techniques can be migrated to components, rather than inventing new replication techniques for components. In the spirit of component middleware, the responsibility of replication management is delegated to the container (*container managed replication*). Three replication approaches are considered, beginning with a simple approach into which incrementally incorporate additional sophistication.

The first approach is the *State replication with single application server*. In this approach, the database is replicated and its failures can be masked, but not the application server's ones. The approach to database replication leaves the container, transaction managers internal to databases and the transaction manager of the application server undisturbed. In order to do it, *proxy resource adaptors* have been introduced. They reissue the operations arriving from the transaction manager and the container to all replica databases via their resource adaptors. Hence, neither the transaction manager nor the containers can to distinguish the proxy resource adaptors and the real resource adaptors communicating with the real JDBMS. Moreover the APIs exposed by the proxies are the same as the real adaptors, and the main task is to direct, to the appropriate database replica, requests of access by the containers and requests of transaction by the transaction manager. With this mechanism, if a transaction fails during the execution, the transaction manager does not roll it back, and the request is issued to the proper replica.

To ensure database replica states remain mutually consistent, the proxy resource adaptor maintains the receive ordering of operation invocations when redirect them to the appropriate resource adaptor replicas. This guarantees that each resource adaptor replica receives operations in the same order, thus preserving consistent locking of resources across resource manager replicas.

In the second approach, *State replication with clustered application server*, the database is replicated and multiple application servers are used for load sharing the total number of transactions in the system. In fact, to ensure a transaction manager does not present a bottle neck in the system and a single point of failure, application servers are replicated complete with the transaction managers. In this case the previous architecture, with resource adaptor proxies masking resource adaptor replicas, is maintained. The proxies, belonging to different application servers, are managed using a group communication system that supports the abstraction of a process group.

In the last approach, *State and computation replication with clustered application servers*, the database is replicated and instances of beans are replicated on the cluster of application servers. For masking application server failures independently of state replication, the containers must be replicated on distinct application servers (with distinct transaction managers) and the states of EJBs in container replicas and transactional states within the respective transaction managers must be mutually consistent. This last case is more complex than the other two because it

requires changes to the application servers and consequently to the containers and the related services.

In conclusion, referring to the first replication technique, that has been implemented, we shall integrate the new mechanism in middleware platform at different stages.

First of all, the Deployer must specify the reference names of the resource adaptor proxies inside the EAR file mentioned earlier and precisely in the deployment descriptors of each component. Thus, all the requests to databases are first captured by these new entities. Then, at the deployment time, the CS enabled must configure the environment and, specifically here, it must allocate the necessary number of database with their replicas according to the SLS. We may think to have a DNS system use to specify which databases are labeled as replicas.

At this moment the CS instructs the resource adaptor proxies to enable requests directed to the booked resource adaptors which interface with the JDBMS. The replicas allocated are then part of the pool of the resources that must be used during the execution of the application components. For these reasons, the replicas are being monitored by the Controller Service (CTRL) whose activity is described in more details in the next Section.

4.5 Controller Service implementation

The CTRL is the middleware service responsible for the adaptive behavior of the architecture. Modeling an adaptive behavior requires the use of monitors that check the resource states as well as the performance levels to be achieved, as specified within the SLAs. The monitor functionality is implemented with the support of a collection of Sensors, as previously explained in paragraph 3.5.3. Sensors send the performance values to the RS, which collect them into reports used for producing log files, statistical analysis, as well as irrefutable proofs in case of disagreements. The data related to the hosted application are sent by the RS to the CTRL, which compares the QoS achieved with the QoS agreed, given by the CS.

During the execution of the application, if the CTRL detects any difference between the values above, it adopts an adaptive strategy to enforce the required service. The applied strategy provokes a reconfiguration of the environment that may trigger a *re-negotiation* involving the CS.

Otherwise, if no adaptive strategies can be applied, *callback mechanisms* can be necessary in order to allow application to deploy application specific adaptation mechanisms (e.g., handlers at the application layer).

The pseudo-code shown in Figure 21 describes the protocol used by the CTRL.

```
Controller Service

ENABLE_CTRL(agreed_QoS, CS_id)
begin
/*checks the hosted application */
   /*asks the Reporting Service for data */
/*compare the data obtained with agreed_QoS */
if (ok) then /* continue to monitor */
else if (between the warning point and limit point (Fig. 10)) then
        begin
        /* apply  adaptive strategy */
        /* calls Configuration Service (CS_id) for the discovery, negotiation and reservation */
        /* again as indicated in the strategy */
        end;

 else if (breaching point (Fig. 10)) then
      begin
      /* no adaptive strategy to apply */
      /* callback to the application layer using the Coordinator */
      end;
 end.
```

Fig. 21: Controller Service protocol

5 Concluding remarks

In this Report we have described the "interim" (see below) TAPAS architecture for application hosting we propose, and examined possible implementations of this architecture within two specific open source application servers; namely, Jboss [JBOSS 2003] and JOnAS [JONAS 2003].

In essence, our architecture extends these application servers with two principal services that provide support to QoS-aware containers. Specifically, this architecture i) derives from the SLA [Beckman *et al*. 2002] the non functional properties an application may require from its run time environment (i.e., its container), ii) configures an appropriate run time environment that holds these properties in order to execute the application, iii) monitors the execution of the application in order to assess whether these non functional properties are maintained by the application run time environment, and, finally, iv) in order to provide the application with a stable run time environment, reconfigures this environment on the fly if events occur which may conflict with the required properties.

As discussed in the TAPAS project proposal [TAPAS 2002], working prototypes of this architecture will be developed in the next 18 months of this project. These prototypes will implement containers enriched with the services described in this Report, and will provide us with valuable insight as to possible revisions that may be required to the TAPAS architecture we have proposed in this Report (hence the use of the term "interim", above).

6 References

[Baochun 2000] Li Baochun *Agilos: A Middleware Control Architecture for Application-Aware Quality of Service Adaptations*, PhD thesis, University of Illinois at Urbana-Champaign, Illinois 2000.

[Beckman *et al*. 2002] W.Beckman, J.Crowcroft, P.Gevros and M.Oleneva "TAPAS Deliverable D1", University of Cambridge and Adesso AG, *EU IST Project 34069 (TAPAS)*, October 2002.

[Beckman & Oleneva 2002] W. Beckman and M. Oleneva, "ADESSO Application Hosting Requirements", *EU IST Project 34069 (TAPAS)*, Dortmund, June 2002.

[Bergmans *et al*. 2000] l.Bergmans, A. Van Halteren, L. Ferreira Pires, M.Van Sinderen and M. Aksit "A QoS Control Architecture for Object Middleware", *In Proceedings of 7th Intl. Conf. on Interactive Distributed Multimedia Systems and Telecommunication Service*, 2000.

[Bodoff *et al*. 2001] S.Bodoff, D.Green, E. Jendrock, M. Pawlan and B. Stearns *The J2EE Manual*, Sun Microsystems Inc., 2001.

[Bollella & Gosling 2000] G.Bollella and J.Gosling "The Real-Time Specification for Java", *IEEE,* 2000.

[Bollella *et al*. 2000] G.Bollella, S.Furr, D.Hardin, P.Dibble, J.Gosling, M.Turnbull, R.Belliardi *The Real-Time Specification for Java*, Addison-Wesley, 2000.

[Braden *et al*. 1997] Braden R. and Zhang L. et al. "Resource ReSerVation Protocol (RSVP)-Version1 Functional Specification", *IETF RFC 2205,* Sept. 1997, http://www.ietf.org/rfc/rfc2205.txt.

[Braden *et al*. 1994] Braden R., Clark D. and Shenker S. "Integrated Services in the Internet Architecture: An Overview", *IETF RFC 1633*, 1994, http://www.ietf.org/rfc/rfc1633.txt.

[Carson *et al*. 1998] Carson M. et al. "An Architecture for Differentiated Services", *IETF RFC 2475*, December 1998, http://www.ietf.org/rfc/rfc2475.txt.

[Checcet & Marguerite 2002] E. Checcet and J. Marguerite *JONAS v.2.4 Tutorial* Rice University INRIA, France, 2002.

[Cockburn 1997] A.Cockburn "Use Case with Goals", *Journal of Object Oriented Programming*, 1997. Available at http://members.aol.com/acockburn/papers/usecases.htm.

[Cook *et al*. 2002] N. Cook, S.K. Shrivastava and S.M. Wheater, "Distributed Object Middleware to Support Dependable Information Sharing between Organisations", To appear in Proc. IEEE Int. Conf. on Dependable Syst. and Networks (DSN-2002), Bethesda USA, June 2002.

[Eugster *et al*. 2001] P.Th.Eugster, R.Guerraoui and C.H. Damm "On Objects and Events", *In Proceedings for OOPSLA*, Florida, October 2001.

[Ezhilchelvan *et al*. 2001] P.D. Ezhilchelvan, S.K. Shrivastava and M.C. Little "A Model and Architecture for Conducting Hierarchically Structured Auctions", *IEEE ISORC 01*, Magdeburg, May 2001.

[Fay *et al*. 1997] Victor Fay et al "Real-Time CORBA", *In Proceedings of the 3rd IEEE Real-Time Technology and Applications Symposium*, Montreal, Canada, June 1997.

[Ferrari 1998] D.Ferrari “The Tenet Experience and the Design of Protocols for Integrated-services Networks”, *ACM Multimedia Systems*, Vol 6., 1998, 179-185.

[Ferrari G. 2002] G. Ferrari “Applying Feedback Control to QoS Management”, *7th Cabernet Radical Workshop,* Bertinoro (FC), October 2002.

[Firesmith 1995] D. G. Firesmith, "Use Cases: the Pros and Cons", Knowledge Systems Corporation (http://www.ksc.com/article7.htm).

[Fowler *et al*. 1997] M.Fowler and K. Scott *UML Distilled*, Addison-Wesley Ed., 1997.

[Ghini 2002] V. Ghini *QoS-Adaptive Middleware Services*, PhD Thesis in Computer Science, University of Bologna, February 2002.

[Gill *et al*. 1999] C.D. Gill, D. L. Levine, F. Kuhns, D. C. Schmidt, J. P.Loyall and R. E.Schantz “Applying Adaptive Middleware to Manage End-to-End QoS for Next-Generation Distributed Applications”, *In Proceedings of the 1st IEEE International Workshop on Real-Time Mission-Critical Systems: Grand Challenge Problems*, 1999.

[ITU/ISO 1995] ITU/ISO Geneva “Information Technology - Open Distributed Processing-Reference Model: Fondations" *ITU-T Recommandation X.902 (Nov. 1995). Also ISO/IEC International Standard 10746.2.*

[Jacobson 1987] I. Jacobson, Object-Oriented Development in an Industrial Environment, *Proceedings of OOPSLA´87, SIGPLAN Notices*, Vol. 22, No. 12, pages 183-191, 1987.

[Jacobson 1992] I.Jacobson *Object Oriented Software Engineering – A Use Case Driven Approach*, Addison – Wesley Ed., 1992.

[JBOSS 2003] http://www.jboss.org/

[JONAS 2003] http://www.objectweb.org/jonas/current/doc/JOnASWP.html

[Kakadia 2001] D.Kakadia “Tech. Concepts: Enterprise QoS Policy Based Systems & Network Management”, *Solaris Bandwidth Manager Software White Paper,* Sun Microsystems, 2001.

[Lamanna *et al*. 2003] D.Lamanna, J.Skene and W.Emmerich “SLAng: A Language for Defining Service Level Agreements” Draft, London, January 2003.

[Lee 1994] W. Lee, ”How to adapt OO development methods in a software development organization — a case study”,in Addendum to the proceedings on Object-oriented programming systems, languages, and applications, ACM Press, Portland, Oregon, United States, 1994, pp. 19—24.

[Lodi 2002] G. Lodi, " End-to-end QoS-aware Middleware Services", *7th Cabernet Radical Workshop*, Bertinoro (FC), Italy, 13-16 Oct. 2002.

[Loyall *et al*. 1998] Joseph P.Loyall, Richard E. Schantz, John A. Zinky, David E. Bakken “Specifying and Measuring Quality of Service in Distributed Object Systems”, *Published on the Proceedings of ISORC'98*, 20-22 April 1998 in Kyoto, Japan.

[Malan & Bredemeyer 1999] R.Malan and D. Bredemeyer "Functional Requirements and Use Cases", 1999. Available at http://www.bredemeyer.com/usecases.htm

[Menasce' *et al.* 2001] D.Menasce', D. Barbara and R.Dodge "Preserving QoS of E-commerce sites Through Self-Tuning: A Performance Model Approach", *In Proceedings of ACM Conference of E-Commerce,* Tampa, October 2001.

[Meyer 1998] B. Meyer, "OOSC2: The Use Case Principle", Eiffel Liberty Journal, Vol.1. No. 2, 1998 (http://www.elj.com/elj/v1/n2/bm/use-cases)

[Miklos 2002] Zaltan Miklos "Towards an Access Control Mechanism for Wide-area Publish/Subscribe System", *In Proceedings of the International Workshop on Event based Systems*, July 2002.

[Morgan *et al.*, 2002] G. Morgan, A. I. Kistijantoro, S. K. Shrivastava and M.C. Little, "Component Replication in Distributed Systems: a Case study using Enterprise Java Beans", The University of Newcastle upon Tyne, Draft Dec. 2002.

[Nahrstedt & Smith 1995] Klara Nahrstedt and Jonathan M. Smith "The QoS Broker", *IEEE Multimedia*, 2(1), 1995.

[Nobile *et al.* 1997] B.Nobile, M.Satyanarayanan, D.Narayanan, J.Tilton, J.Flinn and K.Walker "Agile Application-Aware Adaptation for Mobility", *In Proceedings of the 16th ACM Symposium on Operating Systems and Principles,* Saint-Malo, France, October 1997.

[Ogush *et al.* 2000] M.Ogush, D.Coleman and D. Beringer "A Template for Documenting Software and Firmware Architectures", Hewlett-Packard Product Generation Solutions, 2000.

[Robertson 1999] S. and J. Robertson *Mastering the Requirements Process*, Addison-Wesley Ed., 1999.

[Rosenberg *et al.* 1999] D.Rosenberg and K.Scott *Use Case Driven Object Modelling With UML: A Pratical Approach* Addison-Wesley Ed., 1999.

[Satyanarayanan 1996] M.Satyanarayanan "Fundamental Challenges in Mobile Computing", *In Proceedings of the 15th ACM Symposium on Principles of Distributed Computing,* May 1996.

[Schmidt *et al.* 1997] D.C.Schmidt, D.L. Levine and S.Mungee "The Design of the TAO Real-Time Object Request Broker'", *Computer Comm. J.*, Summer 1997.

[Schmidt & Kuhns 2000] Douglas C. Schmidt and Fred Kuhns "An Overview of the Real-time CORBA Specification", *Appeared in IEEE Computer special issue on Object-Oriented Real-Time Distributed Computing*, June 2000.

[Shannon 2001] B.Shannon *Java 2 Platform Enterprise Edition v.1.3*, Sun Microsystem Inc., 2001.

[Stark & Jboss Group 2002] Scott Stark and the Jboss Group *JBoss Administration and Development Second Edition*, Atlanta, USA, November 10 2002, Jboss Version 3.0.4.

[TAPAS 2002] “Trusted and QoS-Aware Provision of Application Services (TAPAS) – Annex 1 – Description of Work”, IST Programme Project Proposal IST-2001-34069, 2002.

[Vanegas et al.1998] Vanegas R, Zinky JA, Loyall JP, Karr DA, Schantz RE, Bakken DE, "QuO's Runtime Support for Quality of Service in Distributed Objects", in Proc. IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98), September 1998.

[Verma 2000] H.Verma “UML and Use Cases for Object Oriented Analysis”, Lecture at MIT– E-Commerce Architecture Project, Boston 2000.

[Vogel *et al*. 1995] Vogel A., Kerhervé B., Von Bochmann G. and Gecsei J. “Distributed Multimedia QoS: A Survery". *IEEE Multimedia*, 2(2):10-19, 1995.

[Zhang *et al*. 2002] R.Zhang, C.Lu, T.F. Abdelzaher and J.A. Stankovic “ControlWare: A Middleware Architecture for Feedback Control of Software Performance”, *International Conference on Distributed Computing Systems*, July 2002.

[Zinky *et al*. 1997] John A. Zinky, David E. Bakken and Richard D. Schantz “Architectural Support for Quality of Service for CORBA Objects", *Theory and Pratice of Object Systems*, vol. 3, no. 1, 1997.