Integration of QoS Facilities into Component Container Architectures

Miguel A. de Miguel
E.T.S.I. Telecommunicación
Ciudad Universitaria, 28040 Madrid Spain
+34 913 367 366 560
mmiguel@dit.upm.es

Abstract

Component-based infrastructures provide support for the development and execution of component-based systems. However, they have limitations in their application in real-time and reliable systems, because they do not integrate facilities to support these types of problems and include limitations of predictability and dependability. These infrastructures are designed to provide support to financial and e-commerce applications; they integrate basic services like the transaction, persistence, security, and events. These are useful services in real-time and telecom applications, but these applications require other types of services to provide predictability and reliability. We introduce some practical solutions to integrate QoS (Quality of Service) services in the component infrastructures and the results that their business components can expect.

Keywords
Real-time Distributed Component Systems, Real-time EJB, non-functional component infrastructures, predictability in distributed Java applications.

1. Introduction

Nowadays, the following three statements are experimented by all designers, developers, and assemblers of distributed applications:

1. In general, non-functional and technical parts of an application (e.g., transactions, dependability) are more difficult to design and to implement than business ones. In addition, most of the time, business parts are worth more than non-functional parts. This is why developers should focus on functional parts.
2. Non-functional and technical parts change more frequently than functional ones. Developers have a go at building stable and multi-domain software components while the software infrastructure they use keeps evolving and is thus heterogeneous. This can be called “infrastructure dependency problem”. A direct consequence is that several providers will provide non-functional parts.
3. Non-functional and technical implementations depend on an “implicit software infrastructure” (e.g., middleware, middleware services, operating system). In some cases, existing architectural patterns such as n-tiers become first-class elements in software architectures with software component frameworks.

These three statements lead to the following rule of thumb: Separate functional and non-functional parts to attain better time to market and better future evolution. These are important objectives of middleware and software component frameworks such as Microsoft COM+ [14] and .NET [9], Sun EJB (Enterprise Java Beans) [7], and OMG CCM (CORBA Component Model) [13]. They integrate services in the container structures and run-time support thereby limiting the scope of infrastructure dependencies. Middleware (e.g., Java RMI) permits to divide an application into distributed parts. Middleware can interoperate and provide non-functional parts. Software component frameworks (e.g., EJB) introduce concepts to ease the assembly of business components together and the usage of middleware services. Among the most important novelties are introduction of required interfaces, attribute-based programming, combination of middleware services, packaging, and assembly. More precisely, attribute-based programming provides a powerful and easy way to bind services to software components without affecting their internals. It allows for example semi-transparent configuration of middleware services in a declarative way (possibly by an administrator). Attribute-based programming is done in EJB and CCM technology through XML configuration files associated to software component. In .NET, the attributes can be associated to the software components directly in the source files and are then integrated within the byte-code as meta-data.

The goal of our work was to make an exploratory research into solutions for the extension of component models to integrate real-time and telecom services. This
adaptation requires the integration of these services in the architecture of component containers, and identifies infrastructure-independent interfaces and component models to provide the access to these services. The more specific objective of this exploration is to identify solutions for the integration of QoS IP services [4,20] (resource reservation services) in the component container architecture, and study the container architecture to make it adaptable to other types of real-time and fault-tolerant services (e.g. scheduling analysis, adaptation to ORB real-time services, object replication and faults detections, replicated and real-time communication mechanism, adaptation to ORB fault-tolerant services)[19]. The integration of QoS services into middleware has been identified as a future challenge for the middleware infrastructures [16,11]. We have based our studies in some prototypes, where we have adapted current implementations of EJB and RMI to introduce a new component type that will support business component with QoS requirements (specially jitters and latency temporal requirements). Java RMI enables the deployment of Java technology-based applications, in which the methods of remote Java objects can be invoked from other Java virtual machines, possibly on different hosts. RMI uses object serialization to marshal and unmarshal parameters and does not truncate types, supporting true object-oriented polymorphism. The EJB architecture is a component architecture for the development and deployment of component-based distributed business applications. It provides support for development, deployment, configuration, assembly, execution and administration of business components.

Section 2 introduces the new model of component that we have developed. Section 3 introduces some experimental examples and benchmarks, and Section 4 includes some conclusions.

2. QoS IP Component Type

DiffServ and IntServ [4] are examples of standards for the extension of Internet, to support real-time as well as the current non-real-time service of IP. Both standards integrate the Resource ReserVation Protocol (RSVP) [20]. In general, a RSVP reservation request specifies the amount of resources to be reserved for all, or some subset of the packets in a particular IP session. The resources reserved include buffers, bandwidth and jitter. The reservation is done in the nodes of the session’s path that support RSVP.

In this section we introduce our real-time component type model [6], which supports the execution of business components that provides a predictable response time, based on reservation services. The solutions introduced in [8] provide the framework to introduce QoS IP into Java programs. But the process will become more complex when multiple components appear, the negotiation algorithms are complex, and the QoS provided depends on the QoS provided by other servers.

The model that we present is based on EJB [12] and RMI [18] specifications. EJB 1.1 specification introduces some component types like session and entity that CCM includes too. The new type defines semantics and configuration attributes based on the QoS services equivalent to the semantic of session and entity types based on transaction and security services. The QoSBean component type semantic is supported by:

1. External Interfaces. The external interface of the component includes the Business interface that includes business methods of the interface, the Home interface that includes the factory methods of the component and the EJBQoSObject interface that is a common interface for all QoSBean components. This interface includes methods for the negotiation of QoS provided by the component, and other general methods.

2. Component Descriptor. This XML file includes the identification of the component type and its attribute values. QoSBean includes attributes for the identification of the default negotiation algorithm, the exclusive or non-exclusive execution of business methods and the levels of QoS provided (QoS Guarantee, Control Load, and Best Effort).

3. Container. The container supports the type of component specified in the Component Descriptor, for the Business interface. It guarantees the exclusion or non-exclusion of business methods, it accesses the QoS services to provide the QoS support of the component, it implements the default negotiation algorithm and implements the services provided to the business component. The Generator of Container has as inputs the Component Descriptor and the Business interface, and generates automatically the container of this component.

4. QoSContext. This interface includes the internal services provided by the container to the business component. It provides methods to implement customized negotiation algorithms and methods to configure some negotiation algorithms (e.g. amount of bytes required for the serialization of method invocation and return) and other general component services.

5. QoSNegotiation and QoSBean. The business component must implement the QoSBean interface and can implement the interface QoSNegotiation to do the customization of the negotiation process. QoSBean includes general purpose methods (e.g. provide the reference to the context and notify reservation events like errors and correct reservations). When the component implement QoSNegotiation, the container does not provide the negotiation support, delegating it to the business component.
The algorithm computes the reservation required.

In our model, client and component must agree on the load that the component can support and the amount of resources that they will use in their communications. A client must negotiate with the component the number of invocations per second that it will invoke each component method. The component can do the reservation that the client requests, it can reject it or it can provide a reservation with a lower bandwidth capacity (the number of invocations accepted will be less than the number required). A problem, is that often, the component will invoke other component methods to service the client invocation, and it will require the negotiation with its new servers to guarantee the client’s request. In this case the component requires the client’s information before it can negotiate with its servers. The server can use two strategies to negotiate with its servers, it can compute global resources for all its clients’ invocations and make a single reservation, or it can make specific reservations for each client. The negotiation process depends on the algorithms to be used [1,15]. We have developed and prototyped two simple negotiation algorithms:

- **Direct Reservation.** In a first algorithm the server provides the negotiation requested when there are enough resources available. If there are not, the negotiation is rejected. This algorithm uses two methods for the specification of load requested: i) ‘bare bandwidth’, the negotiation parameters include the bandwidth reservation specification of invocation and return flows, and this bandwidths will be used for all method invocation from this client to this component, and ii) ‘invocation description’, the client specifies the number of invocations per second for each component method, and the algorithm computes the reservation required.

- **Reservation with Private References.** In this algorithm the client-server component specifies its server components, the methods of its servers that invokes for each of its methods, and the amount of invocations per method. The algorithm associates a private remote reference to each component server for each client in the negotiation process, computes the invocation frequency of each server component, and negotiates with the servers before to make the reservation that the client requests. If in any reservation process there are not enough resources the reservation is rejected and this frees the reservations done in the negotiation process. The algorithm establishes a private reservation for each component server, and will use the private reservation, when a new invocation arrives. This is supported automatically by the container, which updates the remote reference to be used when a new invocation arrives, before delegating the invocation to the business method. This algorithm can create race conditions when we execute two reservations in parallel and two reservations compete for the bandwidth of different interfaces, none of the reservation is admitted but the resources available would be enough for one of them.

### 2.1 QoS IP Component Type Implementation

The model proposed identifies interfaces and configuration attributes. We have implemented the model in a prototype based on Jonas 2.1.1 [10]. Jonas is a public domain implementation of EJB 1.1, which provides most of the tools required for a basic EJB software development. The prototype can be executed in Solaris 2.6 and Windows 2000. EJB identifies different roles in the EJB-based development. Most of the services provided in this prototype are part of role *EJB Container Provider*. This role provides: i) the deployment tools necessary for the deployment of enterprise beans, ii) the runtime support for the deployed enterprise beans’ instances. The container runtime provides the deployed enterprise beans with QoS management, and other services that are generally required as part of a manageable server platform. The container provider insulates the enterprise bean from the specifics of an underlying EJB server.

Figure 2 represents the main classes and interfaces that will be loaded when the EJB server receives a remote invocation to access the new QoSBean component. These classes and interface are part of different tools:

**QoS Basic Services.** We have defined a new layer in Java-RMI to provide QoS services and we have integrated these services into RMI jdk1.2 library [8]. We have developed two implementations with a common interface (*reservation_API* for Solaris2.6 and Windows 2000).

**XML Analyzer.** We have modified the XML DTD of EJB1.1 component descriptor file to introduce the component type QoSBean, and its configuration attributes. A new library supports the new version of XML files.

**QoSBean Container Library.** Two main classes (*QoSContextImpl and JQoSBean*) implement most
of the general services of QoSBean container. The generator of container generates the specific classes of the container for the specific business methods and component attributes.

**Generator of Container.** The inputs of this tool are the component descriptor and the business interface. It generates two container classes (CompQoSIDRemote and CompQoSIDremote) using rmic. The generator updates the rmic skeleton class to provide some of the container services based on interception techniques. The component descriptor file includes references to the business class and interface (ComponentQoSBean and CompQoSID), and the component server will load the container and business classes.

3. Execution Experiments

The examples that we are going to introduce are executed in two environments. The environment of Solaris 2.6 is two machines and one switch of 3Gb; the first is a Sun Ultra 1, 188MB of memory, 1 CPU UltraSparc, and one 10Mb/s Ethernet Interface; the second is a Sun Ultra 10, 248MB of memory, 1 CPU UltraSparc-II at 269Mhz, and one 10Mb/s Ethernet Interface. The configuration files of RSVP daemons divide the interface networks bandwidth in 70% (7Mb/s) for the reservation sessions and 30% (3Mb/s) for non-reservation sessions (the default session). The C compiler is gcc 2.8.1, and J2SE is jdk1.2.2. The RSVP implementation used is RSVP0.5.2 [17]; we have modified this version to avoid the bandwidth borrow between reservations. The environment of Windows 2000 tests is two machines and one switch of 3Gb; the first includes one Pentium III 600Mhz, Windows 2000 Professional, 260 MB main memory, 3Com 3C918 Fast Ethernet 100 Mb/s; the second includes 2 Pentium III 600Mhz, Windows 2000 Advanced Server, 260 MB, 3Com 3C920 Fast Ethernet 100 Mb/s. The C compiler is Visual Studio .NET 7.0, J2SE is jdk1.2.2, and the QoS support is based on GQoS C library [2,3].

![Diagram](image_url)

**Figure 2. QoSBean component model implementation.**

3.1 Control Load Reservations

This is a Solaris example with two types of client, a client that makes a reservation of 200KB and invokes the server method ten times with an argument of 20KB. Another client has the same behavior but without reservation. Both clients are executed in the same JVM. This test includes two executions, in the first we execute the client with reservation alone, and in the second, we execute the client with reservation and the client without reservation together. The behavior of the clients is:

```java
date0 = current date ()
for second = 0 to second 39
    delay until date0 + second
    start = current date ()
    for i=1 to 10 do invocation of server (20KB)
    result[second] = current date () – start
print result
```

This means that we are testing the latency of execution of 10 remote invocations. The server method executes nothing. Figure 3 includes the results. The average latency of client with reservation is 0.295sec and the worst-case response is 0.361sec, when it is executed alone. We can compute this result as 0.361 < (10 x 20KB / (7000000bps / 8)) + 10 x 2 x 10msec (this expression includes the two main values: transmission time of data arguments and context switch times, which is 10ms). The average latency of this client, when it is executed concurrently with the other client is 0.496sec and the maximum latency is 0.532sec. The average latency of the client without reservation is 0.70sec (< (10 x 20KB / (3000000 / 8)) + 10 x 2 x 10msec); the maximum is 0.829 sec.

![Graph](image_url)

**Figure 3. Client execution alone, with reservation and without reservation.**
The second execution includes two clients as in the previous example, with the same behavior and data arguments sizes, but in this case the second client makes a reservation of 1KB (its execution really requests 200KB/sec). The average latency of client with reservation 200KB is 0.467 sec and the maximum value 0.657 sec. The average latency of the client with reservation 1KB is 3.438 sec and the maximum 3.832 sec. The last client lost all its deadlines (the response time is more than 1 sec), and cannot execute ten remote method invocations every second. Figure 4 includes the results of this execution.

3.1.1 Analysis of Execution Results

In this example we can see that RSVP can be used to reduce the interference of non-critical flows in the response times of real-time tasks.

In the Figure 4 the component limits the number of invocations of its client with the bandwidth reservation, if it makes a reservation less than the real request of the client, the packet scheduler of the output interface in the client node will delay the output of the invocation packages. This strategy can be used in the component negotiation, when the resources available are limited (e.g. it requires a high quantity of computation time for each method invocation, or it has too much access to a database). This limitation of the invocations not only limits the load of the component, it limit the traffic in the network too.

3.2 QoS Guarantee Reservations

This example has been executed in Windows 2000. It includes three types of clients with the same code. The code of clients is:

```java
start = current date ()
invocation of server method (data args)
result [this second] = current date () - start
print result
```

The business method of the component is empty. In this example the reservation is of type QoS Guarantee. The program executes 24 clients in one JVM and two EJB components in another JVM; the points in the next Figure represent the response time for each remote method invocation. Two clients make reservation of 750000Bytes/sec. They invoke the remote method with an argument of 50000Bytes, the first client invokes the component bean1, and the second invokes bean2. Another two clients make the same invocations with the same size of arguments, the first use bean1 and the second bean2, but without reservations. Noise clients are 10 clients that invoke the method of bean1 and another 10 that invoke the method of bean2, all them without reservation and an argument size of 625000 Bytes. These 20 clients are created at instant 15sec, and they collapse the interface of their node (all together use a bandwidth of 12.5Mbytes).

In the Figure 5 we can see that the execution of the 20 noise clients do not affect the response time of the clients with reservation, except at the instant when the 20 clients are created (this is the consequence of the CPU consumption for the creation of 20 new objects and threads, not because of the remote method invocation). On the other hand, the clients that do not make reservation can vary their response time between 0 ms and 512ms; the jitter of their invocation is very irregular. The average response time of clients with reservation is 65ms. They have a reservation of 750000Bytes/sec and they use 50000Bytes/sec. 50000Bytes/750000Byte per sec = 66ms. In this example, the reservation for the return flow is enough to avoid the impact of the delay of returns for the clients with reservation.

![Figure 4. Two types of reservation.](image)

![Figure 5. Direct communication of clients and component.](image)
3.2.1 Analysis of Execution Results

As we can see, we can use the reservation to limit the response time of method invocations, and the QoS Guarantee provides a more limited transmission time. When the size of the argument is high, and the reservation is larger than this size the time of the method invocation is limited to argument size/reservation. This is not only a guarantee for the client; it is a guarantee for the server too, because as we can see the invocations do not arrive with an invocation time less that 60ms. To evaluate these times we must consider the perturbations because of the context switch and the precision of java clock (10ms).

3.3 Playing Audio Data

This example is based on the class ‘AudioDevice’ of Java library. This class reads periodically audio data from an input stream and then it reproduces this data. If there is not data available the execution is delayed and the audio is reproduced noisily. Another thread (we will name it reader) must write the data in the stream (the data must be available before the AudioDevice thread requests it). In this example, both threads are executed in the same Windows 2000 system, and in the same JVM. The reader thread makes the remote method invocations to read the data located in another computer, which execute the EJB QoSBean component that provides access to remote files. This component provides methods to open a file and read a certain amount of bytes. Reader writes the data into the stream in blocks of 4000Bytes. For the type of audio data that we use, the reproduction of this data block takes around 0.5sec. In this example, the reader start to read a new block of 4000Bytes when it detects that the amount of data in the buffer is no more than 6000Bytes. It stops when it detects that the amount is 10000Bytes (the maximum of audio data in the buffer is around 1.25 seconds). The thread in AudioDevice reads around 4000Bytes with a period of 0.5sec. Smaller quantities have a non-regular period. Each execution of this test is 40sec (320Kbytes). In this example the negotiation is based on invocation description solution. The reader thread has two configuration parameters. The first is the size of the arguments of the remote method invocation. A small argument size implies many more invocations. We have tested 10 argument sizes: 80, 100, 125, 160, 200, 400, 500, 800, 2000, and 4000 bytes. The number of invocations to read 4000Bytes for each size is 50, 40, 32, 25, 20, 10, 8, 5, 2, and 1. The second parameter is the scale of reservation factor that provides the support to make much more reservation than the amount required; bandwidth reservation associated to scale 1 is strictly the amount required to execute the remote invocations n times per second. The number of invocations per second for each argument size is 100, 80, 64, 50, 40, 20, 16, 10, 4, and 2; these numbers of calls provides a global bandwidth of 8000Bytes/sec for all sizes. With scale 2 the reservation will be two times the bandwidth required for scale 1, for scale 5 is 5 times, and for scale 10 is 10 times. ‘invocations_description’ reservation solution includes as reservation parameters the list of methods invoked, the number of invocations per second and a reservation scale factor. In this example the list includes a single method (read), the number of invocations per second, and scale factor that we have presented. The algorithms of reservation included into the service ‘invocations_description’, compute the bandwidth required to transmit the heads of RMI call and return streams, the arguments and the return data, the RMI protocol streams, and the TCP ACK packets. The bandwidth depends on the number of packets to transmit per second and the size of packets. In the example that we present, Window 2000 adds to the reservation requested the size of heads of TCP packets.

The Figure 6 represents the time average required to read 4000Bytes of data for each argument size, and for the four scale factors.

The reservation generated by ‘invocations_description’ is not the same for all argument sizes. The Figure 7 represents the reservation, evaluated with tcmon Windows 2000 tool, for the invocation and the return sessions. We have multiplied by 5 the reservation of the invocation to produce values of the same scale of the return (the return includes the audio data and requires much more reservation). When the argument size is 80 bytes, ‘invocations_description’ services compute the largest invocation reservation. ‘invocation_description’ services compute similar values for all return reservations. The differences are especially important for the invocation flow, because the reservation for the invocation arguments is only 4 bytes per call (the number of bytes to be read). The reservation is decreased monotonically, until we arrive to argument size 500, where new a load arrives because of RMI marshaling implementation. Sizes 2000 and 4000 include the bandwidth required to transmit the TCP ACK packets (this is not required for the others because the period is less than 200ms).
The Figure 8 represents two types of results, i) the response times to read 4000Bytes for argument size 200bytes, with scales 1, 2, 5, and 10, ii) the instants when the ‘AudioDevice’ threads start to read a new block of 4000Bytes, (these instants are represented on top of the bars, and in general it is 500ms the time required to consume 4000Bytes). When the scale is 1, the stream finished practically at the same time that the audio finished to reproduce the last data stream, because of this the quality of the reproduction is not good enough. For the other scales there data is available earlier.

These tests have been executed in parallel with ttcp program (ttcp –l400 –n100000) that creates noise, and the responses are the same when the test is executed alone. ttcp is a tool, which can be used to measure end-to-end network transmission time. ttcp sends a sequence of test packets through a network then reports on the delay experienced by each packet. ttcp detects a bandwidth capacity of 13KB when it is executed alone, and 7812KB when it is executed in parallel with this test with scale 5.

### 3.3.1 Analysis of Execution Results

The results of Figure 6 shows that the service ‘invocations_description’ makes reservations that produce practically the same response time for all arguments sizes, this means that we can get the same response times for different argument sizes and invocation frequencies. We spend around 500ms to read the 4000Bytes for all argument sizes, when the scale is 1. When the scale is 2 the delay is 250ms; when it is 5, 100ms; and when it is 10, 50ms. Figure 8 shows the impact of the scale factors in the response time of method invocations. When the bandwidth reserved is low, the invocation will be shaped longer. The combination of shaper and the traffic control scheduling provides the support to reduce the jitters; the variance of response times is similar for all scales and very limited compared to the non-reservation execution.

### 4. Conclusions

We have shown that the integration of QoS IP services into containers significantly raises the abstraction level of middleware, and consequently of software architectures built onto them. Configuration and usage of middleware services that had to be imperative, global, and interleaved in application code with middleware are done locally through attributes declaration in software component frameworks. These extensions will provide facilities to
integrate non-functional services and infrastructures into container architectures and they will limit the dependencies of technical infrastructures. The support for the business and technical separation should be integrated into development processes.

Some open issues of component infrastructure are that they are still complex to seamlessly integrate into software component frameworks such as QoS handling, fault tolerance, and real-time. The component infrastructures and specially the containers are a good framework to simplify some complex processes related to QoS services (e.g. negotiation of QoS levels, specification of QoS requested, integration of QoS services and middleware services).

5. References


[17] Sun Microsystems. RSVP Sources.


http://www.ietf.org/html.charters/rsvp-charter.html