Container Model Based on RTSJ Services

Ruth Tolosa[1], José P. Mayo[1], Miguel A. de Miguel[1],
M. Teresa Higuera-Toledano[2], and Alejandro Alonso[1]

[1] Department of Telematics Engineering, Technical University of Madrid.
[2] Faculty of Computer Science, Complutense University of Madrid,
Ciudad Universitaria, 28040 Madrid Spain
mmiguel@dit.upm.es

Abstract. The container is a basic structure of J2EE used to reduce the complexity of clients and applicative services. Component-based infrastructures use this type of structures and 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 neither integrate facilities to support these types of problems nor include services of predictability and dependability. RTSJ is a basic framework that includes the basic services for the construction of Java real-time systems. The combination of both Java approaches (Containers and RTSJ) is a good solution to reduce the complexity of real-time Java programs. The integration requires the adaptation of both approaches. In this paper we introduce a new model of component container that integrate the RTSJ services based on a resource reservation model.

1 Introduction

Currently, the development and implementation of real-time systems requires the detailed comprehension of some complex techniques (e.g. scheduling analysis techniques, real-time memory management), and the languages or APIs that support these concepts (e.g. RTSJ [2] and RTCore [10]). The integration of these APIs and techniques with application specific problems increases the complexity of design and implementations of applications and their mantenability.

J2EE uses middle-tier server solutions to reduce the complexity of clients and application services. EJB (Enterprise Java Beans) container is a basic structure to support the J2EE middle-tier architecture [5]. The container is a runtime environment that controls the business implementations and provides them with important system-level services. Since developers do not have to develop these services, they are free to concentrate on the application methods. The containers support the common technical solutions, and isolate dependencies of specific implementations. This approach avoids the problems of incorrect use of RTSJ APIs and simplifies the component. The system-level services that integrate current models of containers do not include real-time facilities such as RTSJ services. Current models of EJB are supported by services such as transaction, persistence, and security.

The model of container that introduces Section 2 and the basic services of RTSJ define a component model equivalent to other EJB object types (session, message, and entity), which address other types of problems. This new type of component gives support to represent conversational interactions that require limited response times and resource consumption. In this paper we introduce some solutions for the integration of RTSJ (Real-Time Specification for Java) APIs in container infrastructures. In this integration the container controls two basic resources that supports RTSJ: CPU and memory. The management of these resources in the container isolates in the component container the problems of concurrency, memory management, predictability and synchronization, which support RTSJ. In some previous works we have designed container models that provide predictability of methods invocations based on network reservation and admission control services [6,7]. This paper includes a container model that executes components in local mode (the system executes in mono-processor mode, as RTSJ) and we use the RTSJ resource services to improve the predictability of application components.

The Section 2 includes the general properties of this component model, Section 3 introduces some practical solution to support this model with RTSJ, Section 4 includes some practical results, Section 5 includes the related work and Section 6 includes the conclusions.

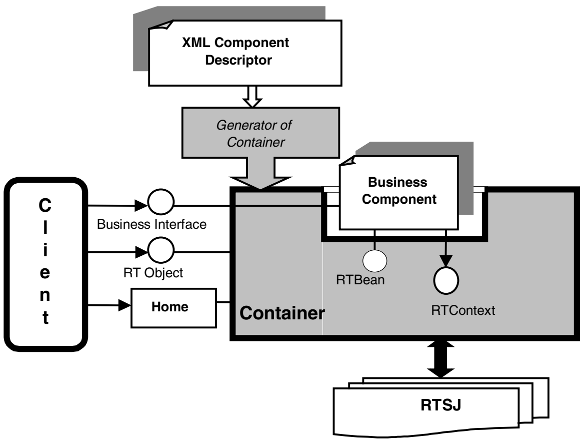


Fig. 1. RTC component model

2 Real-Time Component Model

The *resource-consuming component* is a processing entity that includes a group of concurrent units of execution, which cooperate in the execution of a certain activity and share common budgets. The *budget* is an assigned and guaranteed share of a certain resource. A component has associated: i) *facets* (interfaces provided and synchronously or asynchronously used by clients), and ii) *receptacles* (interfaces synchronously and asynchronously used by this component). A component can negotiate the arrival-pattern and latency for its facets and receptacles. The process of negotiation establishes the temporal properties of component execution. The container implements the negotiation associated to the facets; the container negotiates that there are resources available and reserves the resources. Specific services included in the component's facet support the negotiation facilities and the container intercepts the invocation of these services. The negotiations are expressed in terms of the quality attributes that define the arrival-patterns and latencies of method invocations and memory usage that support the component.

Figure 1 includes the main blocks associated to the component and its development process. `Home` interface and the *Business Interface* are structures defined in EJB. The rest of block are inspired in EJB model, but we adapt them to real-time and RTSJ domains.

1. **External Interfaces**. The external interface of the component includes the `Business Interface` that identifies the business methods of the interface, the `Home` interface that includes the factory methods of the component, and the `RTObject` interface that is a common interface for all RTC (Real-Time Component) components. `RTObject` includes methods for the negotiation and other system services.
2. **Component Descriptor**. This XML file includes the identification of the component type and its attribute values. RTC includes attributes for the identification of the type of reservation that is used and the resources required in the component in general and in its methods specifically, the maximum number of references that the component can support, maximum number of concurrent invocations supported, scale of times used in time expressions, and execution times for applicative methods.
3. **Container**. The container supports the type of component specified in the *Component Descriptor*, for the `Business Interface`. The container implementations use RTSJ to guarantee the allocation of resources for the component, it manages the concurrency in the component, and uses RTSJ for the evaluation of the response time negotiated with clients. The *Generator of Container* has as inputs the component descriptor and the `Business Interface`, and generates automatically the container of this component. The container intercepts any method call to the component; it delegates the applicative methods, but during the execution it and the basic services monitor the method execution.
4. **RTContext**. This interface includes the internal services provided by the container to the business component. It provides methods for the dynamic specification of resources required (memory and execution times) during the component construction, provides information about the CPU available for the current budget, methods that evaluate the proximity of the deadline, methods that return the internal memory available in the component, and other general component services.
5. **RTBean**. The applicative component must implement the `RTBean` interface. In this solution the business component cannot customize the negotiation process as in [6], the component descriptor includes all the information to configure the negotiation. This interface includes methods for the notification of new reservations that can affect the execution of this component.
6. **Home**. The home is a component factory that creates component instances and returns component references. Each component instance can have associated a set of references and a set of clients can use the same reference to communicate with the same component instance.

2.1 Model of Method Invocation, Resource Reservation, and Negotiation

The interface `RTObject` includes a set of services for the negotiation with its clients the arrival-pattern invocation that the component can support (e.g. periodic, aperiodic, group-server), the specific parameters of each pattern (e.g. minimum period, maximum number of concurrent executions), and the allowed latency (e.g. deadline) for the responses. The arrival-pattern is a constraint that the client must achieve, and the latencies are the constraints that the component achieves, when the negotiation contract is done. The container intercepts these invocations and uses the RTSJ services to make the CPU reservation and detect the admission control problems. The containers support two types of reservations that depend on the method to describe the cost of execution in the component. i) The component descriptor includes the execution cost for each business method, and the arrival pattern includes the identification of methods that will be requested. ii) The client only specifies the percent of CPU that the component must reserve for any invocation associated to this reservation. In both cases, the container computes the budget, uses RTSJ services to make the reservation and to evaluate the new schedulability configuration.

The model of reservation could be based on several programming concepts. Three examples are: i) the negotiation process reserves resources for the specific client that makes the negotiation, ii) the negotiation affects to all clients of a component instance, and all clients share the same reservation, and iii) the reservation is associated to component references, and all clients that share the same reference, share the reservations, but different references have different reservations. The `Home` interface includes methods for the construction of new components and to get references to components created. We use the third solution, because it can support the other solutions. In this solution, the interface `Home` creates the references and initializes its reservation (initially they have a best-effort reservation). The container associates a pool of threads to the reference (in some cases this pool includes a single thread, the number and type of thread is included in the component descriptor). A multiple thread configuration allows the concurrent execution of invocations for the same reference. In this case, the reservation is a *group reservation* that limits the CPU consumption of the thread group. The resource manager reloads the cost for the execution for all threads every period. Depending on the number of threads, the method invocation can be blocked because there is no thread available, and it will stay blocked until the end of one method invocation for this reference.

2.2 Model of Memory Management

In our memory model the component has associated two types of memory spaces. One space support the instances handled inside the component (*internal memory*), this space is configured when the component is created. The second set of spaces (*external memory*) support the interchange of information (object instances of method parameters and return object values) with the component's client. The component description includes the default type of *internal memory* that the home factory uses, when the constructor do not specify the memory type. The default external memory is the memory active when the method invocation occurs. `RTObject` includes methods for the execution of methods in specific *external memories*. *Internal* and *external* memories must respect the lifetime rules of RTSJ scoped references, and the lifetime of the component (and the *internal memory*) must be shorter than the lifetime of *external memories* in use.

The container updates the active memory in the method invocation and at the end of the method execution reestablishes the memory active in the invocation. The interface `RTContext` includes operations to create the return instances in the *external memory*. The component must not create references from the external objects to the *internal memory*.

The component descriptor includes information about the maximum memory allocation that have associated each method. This information is reused to compute the allocation time of threads and the allocation time is submitted to admission control. This model supports the local execution of components. This model supports the local execution of components. If the execution were distributed, the unmarshal code could instantiate the serialized objects in the *internal memory*.

2.3 Model of Synchronization

The sequence of execution of a method invocation can be blocked because of several reasons until the end of the method invocation. Different types of reasons are:

1. **Synchronized method.** This type of method avoids the concurrent executions inside the component. Different clients, with the same or different references, cannot execute concurrently the same or different *synchronized* methods for the same instance.
2. **Multiple invocations for the same reference.** A reference classified *periodic* or *aperiodic* has associated a single thread to serve the method invocations. A reference classified *group-server* has associated a limited number of threads. Concurrent invocations for the same reference *periodic* are not allowed. But several clients can use the same *group-server* reference simultaneously. If the number of clients (*c*) that execute concurrent invocations is more than the number of threads in the group (*t*), *c-t* clients will be blocked until the end of method invocations.
3. **Container synchronizations.** The container uses some synchronization structures to maintain the consistency of references and other internal objects. For example to detect that a reference in use is not removed. The execution times for the operations that handle these structures are very short, but priority inversions may extend the blocking time.

3 Implementation of RTC Based on RTSJ

RTSJ includes services for the implementation of model of container introduced in Section 2. The basic services provide support for: i) the resource reservation management (CPU and memory), ii) synchronization and concurrency in the component model, iii) patterns of method invocation, iv) limitation of blocking times and response times. These services make the component response time predictable, and limit the resource consumption for the component.

3.1 Invocation Sequence, Resource Reservation, Admission Control

The predictability of response time of components requires contracting the temporal distribution of method invocation from clients. Depending on the type of invocation pattern and the temporal parameters, we must do the resource reservation. In Section 2 we consider three types of invocations (periodic, aperiodic, group-server) with specific parameters. RTSJ includes classes for the specification of release of threads (`PeriodicParameters`, `AperiodicParameters`, `SporadicParameters`) and the class `ProcessingGroupParameters` can group schedulable objects. The container creates pools of `RealtimeThreads` that serve the invocations for the references. The release parameters of threads depend on the type of invocation release and the temporal parameters that include the negotiation protocols.

In the negotiation process, the new `RealtimeThreads` are included as feasible in the schedulabilty analysis and the class `PriorityScheduler` returns the results for the admission control. The scheduling analysis takes into account the memory management information.The component descriptor includes the worst-case execution times for each application method. It will be used as `cost` parameter of constructor of classes

`PeriodicParameters`, `AperiodicParameters`, `SporadicParameters`.

Problems: Class `MemoryArea` provides methods for the computation of memory remaining in a memory block, but RTSJ does not includes services for the evaluation of CPU available for current budget. This reduces the types of negotiation services that the container can implement (the container can not negotiate based on the amounts of CPU non-reserved, or based on the CPU reservations not consumed). This requires specific resource management services implemented in lowest scheduling levels, or some of hooks in the context switch to compute the CPU consumed.

3.2 Component Memory Management

RTSJ memory areas that support the *internal memory* are `ImmortalMemory` and `LTMemory`. The structure of the component model, based on an interceptor design pattern, allows the interception of all invocations , and the container updates the active memory areas before the business method execution and reestablishes the active memory at the end of the execution. The component factory that implements the container for the instantiation creates a new `LTMemory` instance or configures the `ImmortalMemory` as *internal memory* and this do not change during the execution of the component.

The memory schema allows the reference to the external objects that represent the input parameters, during the execution of the component's methods. And the return objects are copied or created in the *external memory* to make references from the external memories. The containers implement the copy of return objects to external memory, when this is needed (if the *internal memory* is `ImmortalMemory` this is not needed). The component descriptor includes the *maximum allocation memory* (the maximum number of bytes that the method can allocate in one execution) for each method. This value and the temporal distribution of method invocations are used in the computation of `allocationRate` parameter of `MemoryParameters`, which is given on the constructor of `RealtimeThread` and is used for the admission control.

3.3 Component Synchronizations and Blocking Times

The synchronizations for the access to the business component and for the race conditions in the container may block the execution of the invocation sequences. The synchronization protocols that support RTSJ (priority inheritance and ceiling protocol) limit the blocking times and avoid the inversion of priority. Classes `PriorityCeilingEmulation` and `PriorityInheritance` support the protocols and we can compute the worst-case blocking times. The container includes the information about the temporal distribution of invocations and the protocols that are used in the synchronizations.

Problems: RTSJ does not include services for the computation of blocking times or their impact in the feasibility analysis. Classes `RealtimeThread`, `Scheduler`, and `ReleaseParameters` includes information of temporal distribution of threads execution, but they and their subclasses do not include associations with synchronization classes. New classes that extend `Scheduler` or `PriorityScheduler` and `ReleaseParameters` can include new methods to define association with synchronization, and the new `Scheduler` class can take into account the blocking time in the feasibility analysis. Section 2.3 includes a type of synchronization (*multiple invocations for the same reference*) that cannot be computed as blocking time of single data resource as the rest of blocking times. The thread pool is an example of multiple instance data resource. This type of blocking requires specific algorithms for the computation of blocking times; [3] includes algorithms to compute the blocking time in multi-unit object protocols.

4 Practical Results

The execution results that we are going to introduce are based on an implementation of component model introduced in Section 2. This implementation does not use RTSJ but uses the services of a *Resource Manager* [8], for the reservation of CPU. The component, clients and resource manager execute in the operating system pSoS, and the hardware is a CPU for multimedia applications (TriMedia TM1000 card). The *Resource Manager* executes in pSoS OS and provides services for monitoring execution times, control of budgets and admission control. The *Resource Manager* assigns the budgets to clusters of tasks and controls the consumption and reload of budgets. The *Resource Manager* uses pSoS to schedule and monitor the execution of tasks. It includes services for the reservation of CPU and it manages the priority of tasks. It decomposes the priorities in two bands. The tasks that have not consumed its priority budget execute in the high priority band, and when they consume their reservation execute in the low priority band with a best-effort behavior.

The implementation of application interfaces is a set of synthetic operations with fixed CPU consumption (in the examples the components include three methods with execution times of 100, 150 and 200 milliseconds). The containers make the reservation of CPU as response to negotiation request of clients. In following scenarios, the clients make as much invocations as possible. The clients execute in the low priority band. We are going to introduce two execution scenarios: basic reservation operations, and references with cluster of tasks and multiple clients.

4.1 Basic Reservation Operations

This execution includes one component instance and three clients. The clients make the invocation of methods 0 and 1, and negotiate the frequency of invocation of these methods or the bare percent of CPU reservation. The sequence of operations for the clients is included in next Table.

Table 1. Reservation of CPU for Scenario 1

Time	Client	Reference	Reservation
0	0	0	No reservation
0	1	1	50% of CPU
180	2	2	method 0, 2 times per second (20% CPU)
480	2	2	method 0, 4 times per second (40% CPU)
780	1	1	remove the reference 1
1080	0	0	method 1, 4 times per second (60% CPU)
1380	2	2	remove the reference 2
1680	0	0	method 1, 2 times per second (30% CPU)
1980	0	0	remove the references 0 and 2

The container creates a cluster of tasks for each reference. During the negotiation process, it makes the reservation of CPU for each cluster. Figure 2 includes the monitoring result for each cluster that provide the *Resource Manager*. The Figure includes the amount of CPU that they use of their reservation. Reference 0 has a best-effort behavior until it makes a reservation. Reference 1 removes its reservation and therefore its budget. Both references also make other modifications of their reservations. The clusters created for each reference disappear when the client removes the reference.

Figure 3 includes the response time of method invocation for the different references. The unit of axis *y* is 10 milliseconds. Figure 3 (a) includes the reference 0 that vary its response time until it makes a reservation (instant 1080). The response time and its variance are bigger when the total reservation of references 1 and 2 is the 70% and 90%. The reference 1 in Figure 3 (a) has response time less than 200 milliseconds for all the execution (because it executes with a base reservation of 50% of CPU). In Figure 3 (b) we can see two response times higher than 100000 milliseconds, when the reference 2 removes its reservation (because of the preemption of reference 0). And between the instants 1500 and 2000 it has no reservation, but it reduces its response times when the reference 0 reduces the reservation.

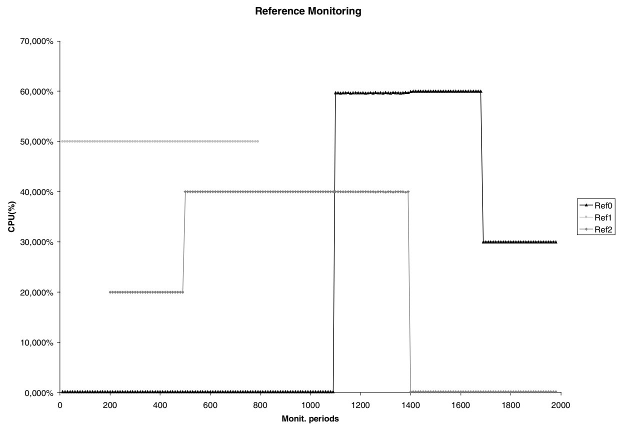
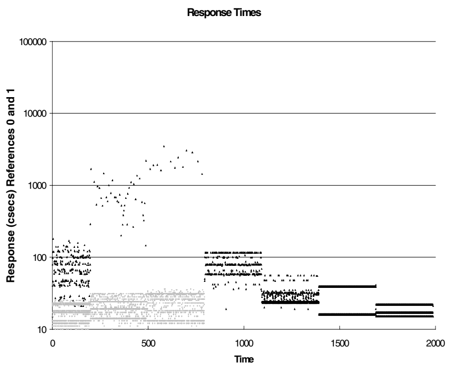
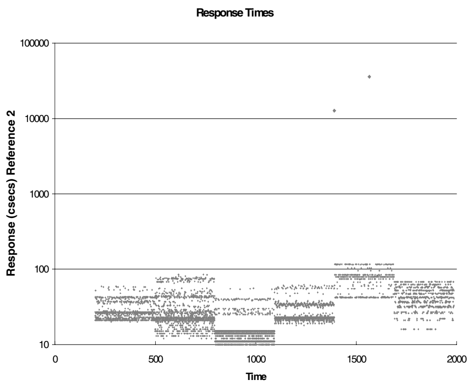


Fig. 2. Reservation of CPU for Scenario 1





(a) (b)

Fig. 3. Response Times for Scenario 1

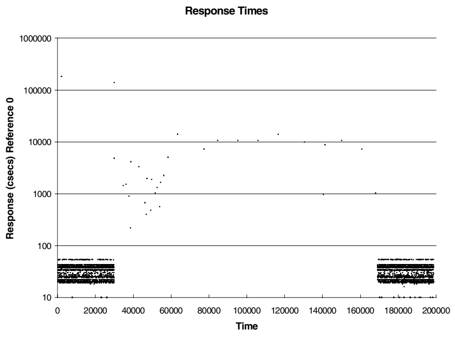
4.2 References with Multiple Clients and Clusters

This scenario creates three references, but there are three clients for each reference. The Table 2 includes the sequence of reservation. All clients compete to make their invocations, and the different reservations produce different response times, for the different groups of clients.

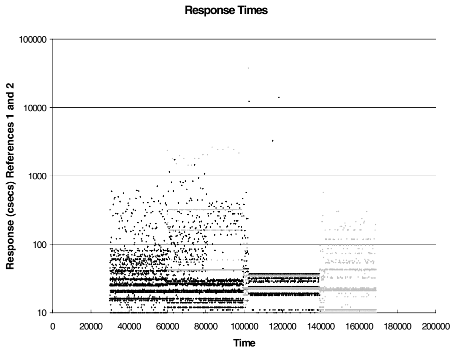
Figure 4 includes the response time for the three references. The unit of axis *y* is 10 milliseconds. Figure 4 (a) includes the response times of reference 0. From instant 300 until instant 1620 the clients that use the reference 0 must compete with the clients that use the reference 1 and 2. Because reference 0 has no reservation and during this interval and the CPU is busy, the number of executions is reduced and their response time is high. In the Figure 4 (b) we can see the different of response time for the references 1 and 2, when the reservation of CPU is 20% and 40%.

Table 2. Reservation of CPU for Scenario 2

Time	Client	Reference	Reservation
0	0, 1, 2	0	No reservation
300	3, 4, 5	1	20% of CPU
600	6, 7, 8	2	method 0, 2 times per second (20% CPU)
960	3, 4, 5	1	40% CPU
960	6, 7, 8	2	method 0, 4 times per second (40% CPU)
1320	3, 4, 5	1	remove the reference 1
1620	6, 7, 8	2	remove the reference 2
1980	0, 1, 2	0	remove the reference 0



(a)



(b)

Fig. 4. Response Times for Scenario 2

5 Related Work

The integration of QoS in component infrastructures is a subject that has a very short history. Most of Component infrastructure standards (EJB 2.0, CCM and .NET) are very recent, and their integration with QoS and Real-Time facilities requires some basic practical improvements (e.g. CCM does not have industrial implementations yet, and open source implementations of .NET has started to appear last months).

Some proposals study the integration of QoS facilities in component models such as CCM (CORBA Component Model) [13,9]. The OMG is currently analyzing propose an RFP (Request for Proposal) for the extension of CCM with basic QoS facilities. The proposal by Wang et al [13] pays special attention to the QoS-enabled location transparency, reflective configuration of component server and container, and the strategies to reconfigure the component server. COACH IST [4] project includes some activities for the integration of QoS facilities in CCM standard. These extensions define generic interfaces to allow negotiation of QoS characteristics between supplier and consumer CORBA components.

Lusceta [1] is a component model (it is not based on industrial component infrastructures) environment based on formal techniques, which can be simulated and analyzed. *Lusceta* provides support for the specification of QoS management, which can be used to synthesize (dynamic) QoS management components. The execution framework is a general QoS-aware reflective middleware. Another component modeling environment is presented in [11]. It proposes solutions for the description of component architectures and for evaluation of response times. This is an architectural environment not supported by execution environments.

[6,7] introduces a solution for the integration of QoS basic services, such as resource reservation and negotiation, in EJB (Enterprise Java Beans). The EJB containers implement some basic negotiation algorithms and isolate the business components from reservation services. The negotiation algorithms implement some basic adaptation process based on the renegotiation of resources and renegotiation with other components.

Schantz et al [12] describe how priority and reservation-based OS and network QoS management mechanisms can be coupled with standards-based, off-the-shelf distributed object computing middleware to better support dynamic distributed real-time applications with end-to-end real-time requirements. They compare two solutions based on priorities and resource reservation for CPU and bandwidth. The reservation solution is based on RSVP and TimeSys Linux RTOS, and the priority-oriented solution is based on Diffserv and TimeSys Linux RTOS.

6 Summary and Discussion

RTSJ provides basic services to support the component model that introduces Section 2. This component model provides high level facilities to make Java components time predictable. The container structure reduces the complexity of applications, and its configured based on attributes. These attributes characterize the temporal behavior and resource consumption of components, and this solution avoids the detailed comprehension of RTSJ complex APIs.

RTSJ can be improved to support models of predictability based on resource reservation. RTSJ include basic services for the evaluation of feasibility of response times, but do not provide information about amount of CPU available, occupation of CPU depending on amounts available, redistribution of non-used CPU (worst case CPU occupation is less than 100%, or the threads do not consume their worst case execution times). Another improvement is the integration of blocking time evaluations in the feasibility schemas. RTSJ includes classes that support time predictable synchronization protocols, but RTSJ do not provide information about blocking times, and there is not associations between these classes and deadline feasibility analysis.

References

1. L. Blair, G. Blair, A. Andersen and T. Jones. "Formal Support for Dynamic QoS Management in the Development of Open Component-based Distributed Systems". *IEE Proceedings Software.* Vol. 148 No. 3. (June 2001).
2. G. Bollella, J. Gosling, B. Brosgol, P. Dibble, S. Furr, D. Hardin, and M. Turnbull. *The Real-Time Specification for Java.* Sun Microsystems, 2000.

3. M. Chen and K. Lin. “A Priority Ceiling Protocol for Multiple-Instance Resource”. Proc. of IEEE Real-Time Systems Symposium, (1991).
4. Coach IST Project. http://coach.objectweb.org
5. L. DeMichiel, L. Yalinalp, and S. Krishnan. *Java 2 Platform Enterprise Edition Specifications, v2.*0. Sun Microsystems, 1999.
6. M. de Miguel, J. Ruiz and M. García, “QoS-Aware Component Frameworks”, Proc. International Workshop on Quality of Service, (May 2002).
7. M. de Miguel “Integration of QoS Facilities into Component Container Architectures”, Proc. 5th IEEE Object-Oriented Real-Time Distributed Computing. IEEE Computer Society, (May 2002).
8. Marisol García-Valls, Alejandro Alonso, José F. Ruiz, amd Ángel Groba “An Architecture for a Quality of Service Resource Manager Middleware for Flexible Multimedia Embedded Systems”, Proc. International Workshop on Software Engineering and Middleware - SEM 2002. Orlando, Florida (2002)
9. Gokhale, D. Schmidt, B. Natarajan and N. Wang, "Applying Model-Integrated Computing to Component Middleware and Enterprise Applications", *The Communications of the ACM Special Issue on Enterprise Components, Service and Business Rules*, Vol. 45, No. 10, (October 2002)
10. J. Consortium Inc. “Core Real-Time Extensions for the Java Platform”. Technical Report. New-Monics Inc. http://www.j-consortium.org. (2000).
11. U. Rastofer and F. Bellosa. “An Approach to Component-based Software Engineering for Distributed Real-Time Systems“. Proc. SCI 2000 Invited Session on Generative and Component-based Software Engineering. IIIS (2000).
12. R. Schantz, J. Loyall, C. Rodrigues, D. Schmidt, Y. Krishnamurthy and I. Pyarali. “Flexible and Adaptive QoS Control for Distributed Real-time and Embedded Middleware”. Proc. Middleware 2003. (June 2003).
13. N. Wang, D. Schmidt, M. Kircher, and. K. Parameswaran. “Adaptative and Reflective Middleware for QoS-Enabled CCM Applications”. *IEEE Distributed Systems Online* Vol 2 No. 5. (July 2001).