An Architecture for Distributed Real-Time Java based on RMI and RTSJ

P. Basanta-Val, M. García-Valls, and I. Estévez-Ayres
Dpto. de Ingeniería de Telemática
Universidad Carlos III de Madrid
Avda. Universidad nº 30, Leganés- Madrid (Spain) 28911
{pbasanta, mvalls, ayres}@it.uc3m.es

Abstract
Today’s trend in real-time systems reveals the necessity of new technologies to easy their development and maintenance. Among others, some interesting alternatives are found in high-level real-time programming languages, better development models, or simple architectures and models. From the perspective of real-time Java, a recent real-time programming language, this paper offers an architecture (and its corresponding Java interfaces) to help the development of an upcoming distributed real-time technology for Java (named DRTSJ). To that end it describes a neutral architecture based on Java’s Remote Method Invocation (RMI) and The Real-time Specification for Java (RTSJ). The empirical evidences included in the paper offer also interesting clues on the performance this technology may deliver.

Terms. RTSJ, DRTSJ, real-time Java, middleware

1. Introduction

By looking some trends in embedded and real-time systems [1] one may conclude that ease of development and maintenance is still an open challenge. Real-time systems, which were both monolithic and isolated pieces of code, are becoming more-and-more networked with important constraints on the amount of resources during its lifetime. This increase in complexity can be reduced from different perspectives: from programming languages, offering high level languages such as it is the case of RTSJ (The Real Time Specification for Java [2]); with new design methodologies, incorporating novel development models such as MDA (The Model Driven Architecture [3]); and from new architectures for distributed real-time systems, such as RTCORBA (The Real Time CORBA [4]). The common goal of these approaches is to deliver a highly cost-effective solution.

The idea of using Java as a real-time technology sounds very appealing. Java offers a portable bytecode model, which may be used to reduce both deployment and maintenance costs remarkably. Furthermore, the size of its programming community and an important number of libraries are synergic factors which should not be disdained in the evaluation of its potentiality. However, its implantation is tough because many built-in mechanisms of Java collide head-on with constraints imposed by real-time systems.

Nowadays, real-time Java has achieved mature solutions for centralized systems and it lacks a similar support for distributed systems. For centralized systems it was already reached an interesting equilibrium point, named RTSJ (The Real-Time Specification for Java). However for distributed systems, such support is still under development. For distributed real-time Java there are two different approaches: one more conservative which bets on the use of already existing technologies (the aforementioned RTCORBA model and RTSJ [5]) and another named DRTSJ (The Distributed Real-Time Specification for Java [6]) which relies on the Java’s RMI (Remote Method Invocation) distribution model.

Both approaches offer remarkable advantages and drawbacks. On the one hand, whereas an RTCORBA mapping may make the most of lessons learned in the past, it also introduces some drawbacks (i.e. the existence of a particular architectural model) which may be not the best solution from the perspective of RTSJ. One the other hand, although DRTSJ is silent about architectural aspects, it still requires some kind of architecture inside to be fully operational. Fortunately, DRTSJ may learn from the previous results on RTCORBA’s predictability [7] to develop its own architecture.

In this context, the current paper proposes and architecture for distributed real-time Java named DREQUIEMI. This architecture contains services specific for RMI like a predictable version of the garbage collector. However, it is general enough to be implemented in other systems and to define extensions to the current RTSJ specification.

The rest of this article gives an overview on the internals of this model. Section 2 introduces the architecture and its primitives. Section 3 deals with details of its API, which will reinforce the ideas exposed along the description of its architecture. Section 4 contains empirical results related to a small evaluation carried out to assess the performance of this kind of infrastructure. Section 5 connects this paper
with other pieces of work. Eventually, the paper ends (in Section 6) drawing conclusions and our ongoing work.

2. The architecture of DREQUIEMI

Before introducing the architecture, it is interesting to take some time to introduce the four abstract goals which were pursued during the design of DREQUIEMI:

- **Alignment with RTSJ’s resource model**
  RTSJ, as a real-time programming language, already has a built-in programming model that allows management of resources in a predictable way. To this end, it introduces some classes such as the MemoryArea or Scheduler; they allow carrying out a predictable management of two important resources: memory and processor. In harmony with this idea, DREQUIEMI maintains and extends this model when necessary.

- **Alignment with RMI’s computational model**
  RMI provides a model for distributed computing based on horizontal and vertical interfaces. They allow the creation of new Java objects which are accessible remotely with built-in support for distributed garbage collection, a connection manager, and a naming service. DREQUIEMI takes them into account defining an architecture which identifies them as key elements.

- **Minimum intrusion rule.**
  From the viewpoint of the architecture developed, the ideal interfaces and models are those that do not introduce an excessive number of new concepts in the model. For this reason the architecture tries to embrace tenets that are currently available in RMI and RTSJ.

- **Be backed by an architecture.**

Merging into a single model two specifications is a hard task since it is not always possible to draw a common goal for both. In the general case, both specifications (RMI and RTSJ) require changes in order to offer a better distribution model. For this reason DREQUIEMI tries to define a higher level model backed by an architecture. This architectural model helped us to identify potential extensions to RTSJ (such as the ExtendedPortal[8] and RealtimeThread++[9]).

2.1. Architecture

Once introduced the goals pursued by DREQUIEMI, it is time to introduce its architecture. As Figure 1 shows, it uses a layered approach which identifies five layers:

- **Resource layer**
  This layer defines the foundations of the model, defining the list of resources that participate in the model. The model takes from real-time Java two key resources: memory and processor and from RMI the network resource. Notice that the current RTSJ specification is silent about different kind of networks, which are not part of the specification.

- **Infrastructure layer**
  These three resources are typically accessed through interfaces (via a RTOS or a RT-JVM) shaping an infrastructure layer. In DREQUIEMI, the access to the infrastructure is given through a set of primitives which are used by programmers to access to low level resources.

  It is important to highlight that to be compliant with the model the current RTSJ requires the inclusion of classes for the network. Fortunately, these classes are already available in standard Java (and in other packages), so they can be added to a new network profile.

- **Distribution layer**
  On top of this layer, the programmer may use a set of common structures useful to build distributed applications. There is one kind of element corresponding to each key resource, namely: ConnectionPool, MemoryAreaPool, and a ThreadPool.

  Each one of these entities represents a complete set of resources, and it may be specialized. Hence, a specific implementation may have different connection pools for different types of networks (e.g. TCP/IP, CAN, TDMA, Ethernet, and so forth).

  Notice that this layer does not provide any specific support to perform remote invocations. This is so because the cost of performing a remote invocation is high and may not be assumed by all distributed applications. For this reason, the model consigns the support for remote invocations to an upper layer.

- **Common service layer.**
  This layer includes two different levels of operation. On the one hand, there are four services: a stub/skeleton
service, a DGC (Distributed Garbage Collection) service, 
a naming service, and eventually a synch/event service.

The first allows carrying out a remote invocation while 
maintaining a certain degree of predictability. The DGC 
service eliminates unreferenced remote objects in a 
predictable way; that is, introducing a limited interference 
on the other tasks of the system. The naming service offers 
a local white page service to the programmers, enabling 
the use of user-friendly names for remote objects. The 
synch/event service is a novel service (not included 
currently in RMI); it allows the establishment of a 
common temporal axis for a distributed application; it is 
based on FTT. Using it, the programs can control 
activation and deactivation of nodes across the network, 
avoiding temporal collisions on the system.

Application.
Lastly, the specific parts of the application functionality 
are found at the uppermost layer, drawn as modules. The 
modules are based on components and promote reuse of 
pieces of other applications.

| Type of middle 
ware | Resource | Primitive |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>_network</td>
<td>_accept _create _close _send _receive</td>
<td></td>
</tr>
<tr>
<td>_memory</td>
<td>_allocate _deallocate</td>
<td></td>
</tr>
<tr>
<td>_concurrent _entity</td>
<td>_create _destroy _setpriority</td>
<td></td>
</tr>
<tr>
<td>_concurrency _limiter</td>
<td>_create _destroy _lock _unlock</td>
<td></td>
</tr>
<tr>
<td>Manager</td>
<td>_set</td>
<td></td>
</tr>
<tr>
<td>Stub</td>
<td>_register _unregister _invoke</td>
<td></td>
</tr>
<tr>
<td>Remoteobject</td>
<td>_register _unregister _invoke</td>
<td></td>
</tr>
<tr>
<td>Naming</td>
<td>_bind _lookup _unbind</td>
<td></td>
</tr>
<tr>
<td>Dgc</td>
<td>_reference _unreference</td>
<td></td>
</tr>
<tr>
<td>Event</td>
<td>_subscribe _unsubscribe _trigger</td>
<td></td>
</tr>
<tr>
<td>Manager</td>
<td>_set</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Current set of primitives in DREQUIEMI

2.2. Current set of primitives

In this model programmers work at two levels: directly 
on the infrastructure middleware (controlling resources 
and simple communication facilities like sockets), or using 
its powerful set of services, running on top of common 
services.

In both cases the communications with the subsystem 
are carried out through primitives (see a complete list at 
Table 1). These primitives are related to certain methods 
of certain classes. For example, the accept method of the 
ServerSocket class may be used to support the accept 
primitive defined by the infrastructure middleware.

The centralized primitives define a set of mechanisms to 
control the three underlying resources (i.e. memory, cpu 
and network). Through its use, it is possible to allocate 
and deallocate memory, concurrent entities, and network 
communications. Moreover, these last two resources (concurrent entities and the network) are modelled in more 
detail with extra support to control the concurrent access 
to resources, the priority of the executing thread, and 
reception and sending of messages through established 
connections. The centralized manager of the system only 
ofers one primitive (set) which allows an application to 
replace the previous running manager with another.

Table 1-bottom contains the set of primitives that give 
access the common services and their management 
algorithms. The subscription and unsubscription to the 
stub/skeleton service is done explicitly with the creation of 
remote objects and stubs (as in RMI). The naming service 
offers three primitives: one to register and deregister 
clients from its internal tables, and another to look them 
up. The DGC service (which is in charge of carrying out 
distributed garbage collections) requires notifications on 
the creation and destruction of remote references hosted in 
remote nodes. The fourth service supports a real-time 
event model which supports publish/subscribe 
communications to send messages, with a trigger 
primitive, to activate remote nodes according to their 
previously declared policies. Same as the centralized 
manager, the distribution layer supports also a global 
manager, which may be changed through a set primitive.

2.3. Management

The architecture described has a very powerful set of 
elements which may be used to configure and tune the 
performance of the entire middleware, locally and 
remotely. They are known as managers and they offer a 
certain degree of independence from the application code, 
increasing the portability of an application. Our current 
implementation incorporates common state-of-the-art real-
time algorithms [9] in its core. More in detail, it gives 
room to the following techniques:

- Preemptive priority-based scheduling.
Same as RTSJ does, DREQUIEMI incorporates support for preemptive priority scheduling, including support to the on-line schedulability analysis, control admission, detection, and recovery policies.

- **Real-time synchronization policies.**
  The utility of real-time synchronization protocols (which are also used by RTSJ) is well-known in the state of the art of the real-time systems. Solutions as the Priority Inheritance Protocol (PIP) and the Priority Ceiling Protocol (PCP) allow reducing remarkably the priority inversion suffered by concurrent entities with higher priorities. In DREQUIEMI, both techniques are used by the middleware to access to shared parts of the middleware, as for example the internal table which stores the list of exported remote objects.

  Apart from controlling the three bare resources, the managers may define and enforce certain application policies to forbid or allow certain operations. For example, one developer may want to define special policies to work with high-integrity distributed systems, forbidding certain actions in the mission phase. Using DREQUIEMI, we may achieve this goal through the definition and implementation of different managers.

3. **Main entities on the API**
The previous section introduced the architecture paying more attention on the underlying model than on programming interfaces. This third section runs in a complementary way, showing how the previous architecture, distribution primitives, and algorithms are mapped to Java classes which are widely based on the specifications of RTSJ and RMI.

Figure 2 introduces the classes of DREQUIEMI’s implementation for distributed real-time Java. Its classes are grouped in three sets. The first of them defines classes for the distribution middleware named resource pools. The second is concerned with the management issues and defines two key classes: one for the default management policies. The uppermost part of the figure shows the classes for the common services of DREQUIEMI’s architecture. In this figure, it may be found the stub/skeleton classes of the stub/skeleton service (RealtimeUnicastRemoteObject and RealtimeRemoteStub), the classes of the synchronization/event service (Synchronization) and those of the distributed garbage collection (DGC) and naming services (Naming).

The rest of this section gives a brief overview of stub/skeleton service related classes, omitting the rest of services. This service is crucial in the implementation because many other services, like the naming service are implemented on top of this service.

A full description synchronization/event service are still available in [14] and [15]. These references include a description of the service, an identification of its main entities and their related Java classes, and an empirical evaluation of the stability (jitter) of the current implementation.

3.1. **Resource Pools: Memory pools, Thread pools and Connection pools**
A pool of resources is an idea that was already included in RTCORBA through the concept of thread pools. They may be understood as common-off-the-shelf patterns which attempt to be independent from an application. One of the easiest one to understand is the pooling pattern. In essence, this pattern consists of allocating a certain amount of resources (i.e. memory, threads or connections) reusing them in subsequent remote invocations. As a result of this practice, the performance of the system gets increased remarkably because the typical initialization overhead simply disappears.

To integrate this pattern harmonically in the proposed architecture, DREQUIEMI’s API offers three Java hierarchies, which may be extended when necessary:

- **ThreadPool**
  RTCORBA already defines a very complete thread pool strategy which allows the use of dynamic and static threads that optionally are assigned to lanes. The current implementation of DREQUIEMI’s thread pool is much simpler. It considers two possibilities: a highly dynamic case and a more static one (useful for example for high integrity systems). In the first case (associated with the DefaultThreadPool class), all threads are of type real-time threads and they are allocated dynamically within the heap memory. In the second case (associated with ImmortalThreadPool), the threads are statically created in immortal memory and are of type no-heap in order to guarantee higher system predictability.
The ORB (Object Request Broker) of RTCORBA enables preallocation of connections and its assignment to a certain band of priorities. This mechanism may be considered as a primitive pooling mechanism. The current implementation of DREQUIEMI goes a step beyond defining this mechanism as central entity of its architecture. It supports two connection pools for TCP/IP networks. The first one (associated with DefaultConnectionPool) allocates all communications dynamically, on demand, to carry out a communication, closing the connection after each remote invocation. The second one (associated with PriorityImmortalConnectionPool) allocates all communication resources within immortal memory and reuses the pre-allocated connections after this initialization, offering higher efficiency.

Lastly, this class is much related to the memory management of RTSJ (i.e. HeapMemory, LTMemory and ImmortalMemory). The key difference among RTSJ’s classes and the three defined by DREQUIEMI (HeapMemoryAreaPool, LTMemoryAreaPool, and ImmortalMemoryPool) is that they are pooled versions of previous structures. As figure 3 shows for an LTMemoryAreaPool instance the class supports (in its constructor) the definition of a minimum number (min) and a maximum number (max) of LTMemory instances that are timely reused in subsequent invocations.

```java
package es.uc3m.it.drequiem.rtrmi.server;
import javax.realtime.*;
import es.uc3m.it.drequiem.rtrmi.*;
import java.rmi.server.RemoteStub;
public class RealtimeUnicastRemoteStub {
    public static void setParameters(
        RemoteStub stub, 
        boolean async, 
        SchedulingParameters sp, 
        ReleaseParameters rp, 
        MemoryParameters mp, 
        ProcessingGroupParameters ppg, 
        ConnectionPool cpool
    )
    
    public static SchedulingParameters getSchedulingParameters(
        RemoteStub stub); 
    public static void setSchedulingParameters(
        RemoteStub stub, 
        SchedulingParameters sp); 
    
    public static ConnectionPool getConnectionPool(
        RemoteStub stub); 
    public static void setConnectionPool(
        RemoteStub stub, 
        ConnectionPool cpool);
}
```

Figure 4: RealtimeUnicastRemoteStub details

Apart from this previous parameterization, a real-time remote stub allows the association of a global behavior to the invocation of void remote methods. When async parameter of the setter methods is set to true, void methods of a remote stub do not wait for a confirmation from the remote server; they just continue with their execution (see more details at [16]).

### 3.2. Real-time stumps and remote objects

The three aforementioned hierarchies of resources are instantiated by the programmer and associated with their real-time remote objects and stubs. This subsection shows how they fit together in the stub/skeleton service. The subsection starts with the real-time remote stub to end with the real-time remote object class. It also details how to perform synchronous remote invocation (which wait for the response from a server) or asynchronous ones (which do not wait for such response). In both cases, all the description is supported by sourcecode of the main classes of the hierarchy.

#### Real-time remote stub

Figure 4 shows details of the code used to support the client side of the real-time remote invocation. In order to avoid changes in the compiler, DREQUIEMI resorts to a glue class. Through static methods, the glue class associates the new real-time aspects with the remote stub class using static getter and setter methods of a generic RealtimeRemoteStub class. These methods allow the parameterization (with sp, rp, mp and ppg) of the remote invocation. In more detail, sp allows configuring scheduling parameters (in our case the priority used in the remote invocation and its transmission to the server); rp defines the operating margins of the application; mp allows the programmer to include information useful for the real-time garbage collector; and ppg is related to the configuration of aperiodic server techniques at the client side of a remote invocation. Lastly the association of a default connection pool is done through the cpool parameter.
Figure 5 show how RMI’s UnicastRemoteObject has been extended to include all the real-time information required to perform a remote invocation. As in the previous case, this was done adding to the previous class management information (sp, rp, mp and ppg) which may be specified in the constructor and modified at runtime through specific getters and setters (see code details in the figure).

```java
package es.uc3m.it.drequiem.rtrmi;
import javax.realtime.*;
public class DefaultPriorityDistributedScheduler
 extends DistributedScheduler {
    public DefaultPriorityDistributedScheduler(
        SchedulingParameters sp,
        ReleaseParameters rp,
        MemoryParameters mp,
        MemoryAreaPool map,
        ThreadPool thp,
        ProcessingGroupParameters ppg )
 public SchedulingParameters getSchedulingParameters();
 public void setSchedulingParameters(SchedulingParameters sp);
    }
```

Figure 5: DefaultPriorityDistributedScheduler details

3.3. Management: Default Distributed Scheduler
The first two subsections introduced the Java’s API of DREQUIEMI and its TCP/IP real-time remote objects and stubs. This subsection details how the programmer (or the system engineer) may configure the system performance through the default distributed scheduler of DREQUIEMI.

Figure 6 gives an overview on the constructor of the default priority scheduler of DREQUIEMI. This class admits a global parameterization for the communication subsystem, as well as the configuration of the distributed garbage collector and the naming service.

The definition of a non-null sp parameter forces the middleware (as in a real-time remote object) to use sp for handling incoming remote requests. The rationale is still valid for memory, release, and processing group parameters.

```java
package es.uc3m.it.drequiem.rtrmi;
public class DefaultPriorityDistributedScheduler 
 extends DistributedScheduler {
    public DefaultPriorityDistributedScheduler(
        SchedulingParameters sp,
        ReleaseParameters rp,
        MemoryParameters mp,
        ProcessingGroupParameters ppg,
        ThreadPool globalthreadpool,
        SchedulingParameters dgcsp,
        MemoryAreaPool dgcmemorypool,
        SchedulingParameters namingsp,
        MemoryAreaPool namingmemorypool,
        ConnectionPool conpool )
    }
```

Figure 6: DefaultPriorityDistributedScheduler details

4. Empirical results
A partial implementation prototype of DREQUIEMI has been developed with a double purpose in mind. On the one hand, it helps validate the architecture, measuring its temporal properties. On the other hand, it is also a valuable platform to test different implementation approaches and to identify implementation bottlenecks. DREQUIEMI is build on top of the source code of RMI for J2ME [18] and JTime [17].

The first experiment carried out quantified the overhead we may expect from a distributed real-time Java in absence of payload from the application. To carry out this experiment, it was measured the time consumed by the main services of DREQUIEMI. After that, it was compared against the direct use of sockets. In both cases, the experiment was executed using the loopback device to minimize the interference of the network (running all tests in a 796 MHz-Athlon XP Laptop). Table 2 summarizes the results obtained after the experiment.
In this table, the first column refers to the cost (in microseconds) of carrying out each remote communication. The second one gives information on the absolute overhead (without socket communication costs) of the communication. The last column works with percentages; it normalizes the overhead of a communication by the one of the socket-based communication.

From the results it may be concluded that middleware, even in cases where there is not data transmission, introduces a non negligible overhead. In the general case, the cost of remote invocation is more than three times higher (4.23) than the one associated to a socket-based application (1.00). This cost is reduced when the remote invocation is asynchronous (2.25). However, in the asynchronous evaluation, the results are unfair because with asynchronous remote invocations the client does not wait for a response from the server, increasing its performance fraudulently. When no application data is transferred to the subscribed nodes, the synchronization service offers a performance similar to the one of a remote invocation. The reason for such coincidence is in the implementation: our current implementation uses remote invocations to transfer the events.

Lastly, the naming and distributed garbage collection services are the most expensive ones in terms of efficiency. Notice that an interaction with the distributed garbage collector introduces a cost that over thirty times (30.27) the one of a socket-based communication. The same rationale is valid for an interaction with the naming service, which increases (44.32) the cost of the communications remarkably. Therefore, an efficient and highly predictable code should avoid the use of these two services.

5. Related-work

Most approaches suggested by real-time Java researchers to the distributed real-time Java issue are constructed on top of well-known specifications. On the one hand approaches such as RTZen [20] are based on the RTCORBA predictability model. On the other hand, another set of approaches are based on the RMI distribution model ([21],[22] and [23]); all of them also built their solutions on the same real-time Java specification: RTSJ. Apart from these specification-based there are other approaches. One of them, proposed by Silva et al. [24] bases its solution on the use of FemtoJava (a native Java processor) and a low level communication interface. The other, proposed by Hilderink et al. [25] bases its approach in the advantages given by the formalism of CSP-channels. The approach described in this paper belongs to the RMI-based family.

6. Conclusions and ongoing work

To build a successful distributed real-time Java technology based on RTSJ and RMI specifications, it is interesting to have a general architecture. This may help understand and control the internal behavior of the entire middleware. The architecture and also the interfaces which were described in this article provide an initial approach to identify the set of services that will be relevant in the future development of a successful distributed real-time Java technology. One of the main lessons learned is that it is not sufficient with use of a theoretical or empirical approach; it is necessary to follow a more integrated approach which considers both issues together.

Our ongoing work evaluates different alternatives to transfer non-functional information among several networked real-time virtual machines (see [26]). Another promising ongoing work is to explore the integration of service composition (see [27]) in distributed real-time Java

References


