# Towards A Framework for Integrating the Real-Time Specification for Java and Java's Remote Method Invocation

Andy Wellings
University of York, andy@cs.york.ac.uk

Ray Clark, and Doug Jensen
The MITRE Corporation, {rkc,jensen}@mitre.org

Doug Wells
The Open Group, dmw-java@contek.com

**Abstract**

*This paper proposes a framework for integrating the Real-Time Specification for Java and Java's Remote Method Invocation. The concepts of real-time remote and distributed real-time remote interfaces are introduced in order to facilitate the design and implementation of real-time and distributed real-time threads that call remote objects.*

## Introduction

The paper discusses the options available for integrating the Real-Time Specification for Java (RTSJ) [1] and Java's Remote Method Invocation (RMI) facility [5]. Although the RTSJ is reasonably well defined, currently no consideration is given as to how its facilities can be used in a distributed environment. Similarly, although RMI is well defined, no consideration has been given as to how/whether it can be used in a real-time environment.

## RMI

Key to developing RMI based system is defining the interfaces to remote objects. RMI requires that all objects, that are to be accessed remotely, provide a remote (by extending the pre-defined interface, `Remote).` Each method defined in this interface must declare that it `"throws RemoteException"`. Thus one of the key design decisions of RMI is that distribution is not completely transparent to the programmer. The location of the remote objects may be transparent, but the fact that remote access may occur is not transparent.

### 1.1 The meaning of time

In standard Java, time is expressed as a number of milliseconds and nanoseconds passed 1st January 1970. Java also has a `Date` class that encapsulates these values. The `Date` class is serializable and consequently objects of this type can be passed through RMI. However, RMI is silent on the issue of the relationship between the clocks on the client and server sites.

### 1.2 Failure semantics

RMI assumes a reliable transport mechanism (TCP). Consequently, it does not need to be concerned with transient communication errors. RMI's handling of node and permanent communication failures is centered on the throwing of a `RemoteException`. A `RemoteException` is thrown if a client makes a remote call and the RMI implementation is either unable to make the call or makes the call but detects a failure before the call has returned. The server is not informed if a client node fails whilst a server is executing a request on its behalf. RMI has *exactly once* semantics in the absence of failures and *at most once* semantics in the presence of failures.

## Minimal Integration between RTSJ and RMI (Level 0 Integration)

Interfaces in Java provide a mechanism for defining a contract between a client and a server. They make no statements about the attributes of any associated object. By implementing an interface, the server is guaranteeing to provide the functionality implied by the interface. *By definition, a Java interface says nothing about the real-time properties of the server. Similarly, a remote interface says nothing about the real-time properties of either the server or the underlying transport system.* Consequently, an object that implements a remote Java interface can be considered to be an object that is executing *without* the requirement of a real-time JVM. Furthermore, the transport protocols that implement the connection between the client and server are not required to be timely, irrespective of whether the client is a real-time client.

One way of viewing the above interpretation of RMI is that the proxy thread on the server (which executes the server methods on behalf of the client) can be viewed as an ordinary Java thread (even if the client is a real-time thread). This, therefore, is a minimal integration between the RTSJ and RMI. Real-time threads can call remote methods but they can expect no timely delivery of the RMI request, and the server and its proxy thread are unaware of any time constraints that the client has. The application programmer must explicitly pass any scheduling or release parameters and require a sympathetic RMI implementation.

The main advantage of a Level 0 integration is that it requires no additions to RMI or the RTSJ. Consequently, Level 0 integration is silent on the relationship between the clocks on the client and the server. It also has the same failure semantics as that of RMI.

## Real-Time RMI (Level 1 Integration)

In keeping with the non-transparent RMI philosophy, to obtain real-time remote communication, Level 1 integration proposes the introduction of a real-time RMI. Key to developing real-time RMI-based system, is defining the interfaces to remote real-time objects (objects that assume that they are executing in a real-time JVM). Real-time RMI requires that all objects that provide a remote interface must indicate so by extending the pre-defined interface `RealtimeRemote.`

```
public interface RealtimeRemote
       extends Remote{};
```

Each method defined explicitly or implicitly in an interface extending `RealtimeRemote` must declare that it

`"throws RemoteException"`.

At the server side, the proxy thread can be viewed as a real-time RTSJ thread that inherits appropriate scheduling and release parameters from the client RTSJ. Where the client is not a RTSJ thread but a standard Java thread, default release and scheduling parameters can be provided.

A `RealtimeRemote` interface indicates that the client can expect the underlying transport of messages and the server-side objects to be aware of any client-side scheduling parameters. However, it offers no guarantee on the type of memory used. A stronger guarantee can be given if the server interface is defined as an extension of the `NoHeapRealtimeRemote` interface. This ensures that the underlying transport of messages and server objects will make no use of the Java heap. The server proxy thread can be viewed as a no-heap RTSJ real-time thread.

### 1.3 Model of time

The level 1 integration of RTSJ and RMI assumes that there is communication of timing constraints between the client and server but that this is transparent to the client and server real-time JVM. For example, as far as the server JVM is concerned, the proxy thread is just another real-time thread whose scheduling and release parameters just happen to be set by the real-time RMI infrastructure. The client and server JVM are, therefore, still independent of one and other. Consequently, there is no relationship between their real-time clocks.

It should be noted, that currently RTSJ timed types are not serializable. This means that any time values passed across the RT RMI must be converted into suitable types and reconstructed at the server site.

### 1.4 Failure semantics

Level 1 integration assumes that nodes (sites) suffer only crash failures. In the absence of failures, Level 1 integration provides exactly once semantics. In the presence of failures, the semantics are at most once. Failures on the server side are presented to the client via remote exceptions. Failure at the client can be either ignored at the server side, or the server can be informed by one of the RTSJ asynchronous communications mechanism.

### 1.5 Limitation of the Real-Time RMI (Level 1) Approach

Although the Level 1 integration requires extensions to RMI, it does not require any extensions to the RTJVM or the RTSJ. However, this results in some limitations that stem from the fact that none of the RTSJ class definitions have remote interfaces. Consequently, they can offer no remote services. Furthermore, with the exception of `AsynchronouslyInterruptedException`, none of the classes implement the `Serializable` interface, and consequently objects of these classes cannot be passed across a remote interface. The next section considers extensions to the RTSJ to increase its distributed real-time programming capabilities.

```
public interface RemoteThread
  extends DistributedRealtimeRemote
{
  public RemoteReleaseParameters getReleaseParameters()
                          throws RemoteException;
  public void setReleaseParameters(
   RemoteReleaseParameters parameters) throws RemoteException;
  /* Similarly for SchedulingParameters */

  public RemoteScheduler getScheduler() throws RemoteException;

  public synchronized void interrupt() throws RemoteException;

  public void start() throws RemoteException,
                      IllegalThreadStateException;
}

public DistributedRealtimeThread extends RealtimeThread
       implements RemoteThread
{
  // Implementation of RemoteThread interface plus

  public static RemoteThread currentRemoteThread()
       throws  NotARemoteThreadException;
}
```

**Figure 1: Distributed Theads**

## Distributed Real-Time Threads (Level 2 Integration)

Following on from from dynamic CORBA[4] and the pioneering work of the Alpha kernel [3] and MK7[6], a distributed real-time thread model is one in which a thread's locus of control can move freely across the distributed system by calling methods in remote objects. Each distributed real-time thread has a unique system-wide identifier and, at any point in time (and in the absence of any faults such as network partition), the thread is eligible for execution (or is suspended) at a single site in the distributed system. This site is the site that is hosting the real-time remote object encapsulating the method that is currently called by the thread. Any remote operations on the distributed thread will affect (via the underlying real-time virtual machines) one or more sites that currently host the distributed thread's execution.

One way of expressing distributed real-time Java threads is to extend the RTSJ's `RealtimeThread` class and to implement a real-time remote interface which defines the remote operations that can be called on the thread. However, to implement distributed threads requires more from the underlying RTJVM and real-time RMI transport protocols than that implied by the discussion in section 0. Consequently, to indicate this, the notions of a distributed real-time JVM and a distributed real-time RMI are introduced. The distributed real-time JVM is a real-time JVM augmented with facilities to support the distributed real-time thread model.

Although it is beyond the scope of this paper to define completely a distributed real-time specification for Java, an example of how the RTSJ `RealtimeThread` can be extended is given. The operations that can be performed on a distributed real-time thread can be classified into two areas:

- *Operations that affect the scheduling of the distributed thread.* These include being able to get and set its release and scheduling parameters.
- *Operations that affect the state of the distributed thread.* These include being able to start a remote thread and being able to interrupt it

Figure 1 shows a possible remote interface and class definition for distributed real-time threads. The RTSJ class definitions for `ReleaseParameters`, `SchedulingParameters` etc will need to be either re-defined (or subclasses created) to implement remote interfaces or new classes created. A new static method allows the current remote thread unique identifier to be found.

### 1.6 The meaning of time

Level 2 integration assumes that there is a cluster of sites that are hosting the distributed real-time application. These sites could, therefore, provide a clock (perhaps separate from the RTSJ real-time clock) that is coordinated across the cluster (that is synchronized to some delta and within a defined accuracy of UTC).

### 1.7 Failure semantics

Sites are assumed to suffer from crash failures only.

In the distributed thread model, the distributed RTJVM is directly supporting the distributed thread semantics. Consequently, when a segment site hosting the distributed thread fails the distributed RT JVM coordinate their responses as follows.

1. If the failed site is the head site, the site hosting the previous segment has a remote exception raised.
2. If the failed site is a segment site other than the origin or head site, the head site implements one of the models proposed for the server site in Level 1 Integration (i.e., ignore, throw an AIE or fire an AE). When the distributed thread tries to return to the failed segment site, the remote exception is raised in the segment site previous to the failed site.
3. If the failed site is the origin site, approach 2 is followed until the distributed thread attempts to return to the origin site. At which point the remote is exception is lost.

When the thread is handling the remote exception or AIE or running the AE handler, details of the failure can be found from the underlying distributed real-time JVM.

## Conclusions

This paper has explored the ways in which the RTSJ can be integrated with Java RMI. An incremental approach has been suggested along the following lines, in order or increased functionality (and increased complexity).

- Real-time Java threads can call remote objects but they can expect no timeliness of message delivery and no inheritance of scheduling parameters — there are no changes to RTSJ and no changes to RMI.
- Real-time Java threads can call real-time remote objects and they can expect timely delivery of messages and inheritance of scheduling parameters; however, they cannot expect to have any distributed thread functionality — there are extensions required to RMI but no extensions required to DRTSJ or the RT JVM.
- Distributed real-time Java threads can call distributed real-time remote objects and they can expect timely delivery of messages, inheritance of scheduling parameters and full distributed thread functionality — the DRTSJ consists of the extensions required to RMI, RTSJ and to the RT JVM.

## Acknowledgement

The research reported in this paper has been performed in the context of the Sun Community Process JSR 50. The authors gratefully acknowledge the help and advice given by members of the JSR 50 Expert Group.

## References

1. G. Bollella et al, *The Real-Time Specification for Java*, Addison Wesley, 2000.
2. A. Burns and A.J. Wellings, *Real-Time Systems and Programming Languages* 3rd Edition, Addison Wesley, 2001.
3. R.K. Clark, E.D. Jensen and F.D. Reynolds, *An Architectural Overview of the Alpha Real-Time Distributed Kernel,* USENIX Workshop on Microkernels and other Kernel Architectures, pp 200-208, 1993
4. OMG, *Dynamic Scheduling Real-Time CORBA 2.0, Joint Final Submission,* OMG Document orbos/2001-04-01
5. Sun Microsystems, *Java Remote Method Invocation Specification*, December 1999.
6. D. Wells, *A Trusted, Scalable, Real-Time Operating System*, Dual-Use Technologies and Applications Conference Proceedings, pp II 262-270, Utica, NY, 1994