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
{rkj, jensen}@mitre.org

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

Abstract

The Distributed Real-Time Specification for Java (DRTSJ) is being developed under Sun's Java Community Process. It is focused on supporting predictable, end-to-end timeliness for sequentially distributed computations (e.g., chains of invocations) in dynamic distributed object systems. This paper reports on an investigation to integrate and extend the existing Real-Time Specification for Java and Java's Remote Method Invocation facility to provide the basis for the DRTSJ.

1 Introduction

The Distributed Real-Time Specification for Java (DRTSJ) is being developed by the JSR-50 Expert Group in Sun's Java Community Process [7]. One of the avenues being explored is to consider the options available for integrating the Real-Time Specification for Java (RTSJ) [2] and Java's Remote Method Invocation (RMI) facility [23].

The paper is structured as follows. In section 2, the motivation for integrating RTSJ and RMI is given. Then in section 3, the key points of RMI are discussed and in section 4, a proposal is made on how minimally to integrate the RTSJ with RMI. The limitations of this proposal are detailed leading to the introduction, in section 5, of the notion of Real-Time RMI and Real-Time Remote Interface along with a discussion of how this can be used with the RTSJ. This approach is then evaluated and several weaknesses are highlighted. Sections 6 and 7 then extend the model to provide full distributed thread functionality and discuss interoperability between the various platforms. Finally section 8 summarizes the proposals and presents conclusions and future work.

2 Distributed Real-Time Systems

There are many different kinds of distributed systems. However, the vast majority of deployed distributed real-time computing systems employ at least one of the following programming models:

- *control flow:* movement of execution point, with or without parameters, among application entities (e.g., remote procedure call (RPC) and remote method invocation (RMI))
- *data flow:* movement of data, without an execution point, among application entities – e.g., publish/subscribe and bulk data transfers
- *networked:* asynchronous or synchronous movement of messages, without an execution point, among application entities (e.g., message passing, IPC)

Other distributed system programming models – e.g., mobile objects, autonomous agents, tuple spaces, web services, etc. – currently are confined almost exclusively to non-real-time systems. For example, most of the approaches that have provided augmented distribution models for Java (for example, Linda [25], JavaSpaces [24], Voyager [20] and JavaParty [17]) have not considered real-time issues [1].

Control flow models are usually designed for multi-node – usually *trans-node* (linearly sequential) behaviors that are synchronous, that is they request remote execution and then wait for a response. When asynchronous response is not needed, one-way invocation can be spawned asynchronously (as in CORBA). An example of the distributed control approach is the distributed thread model. A *distributed thread* is a single thread, with a system-wide ID, that extends and retracts itself sequentially through an arbitrary number of local and remote objects. A *distributed real-time thread* transparently propagates its timeliness properties (and perhaps also resource ownership, transactional context, security attributes, etc.) when its execution point transits object (and perhaps node) boundaries. The distributed thread model appeared first in the Alpha distributed real-time OS kernel [4, 11], and was subsequently incorporated in the kernels of MK7.3 distributed real-time OS [26]; it is the basis of the programming model for OMG's recent Real-Time CORBA 2.0 (Dynamic Scheduling) specification [14]. Realistic non-trivial experimental distributed real-time computing systems were successfully constructed using Alpha [9] and MK [7, 5].

Almost all data flow models, including those for "real-time" publish/subscribe, are more oriented toward maximizing throughput than maintaining end-to-end timeliness properties (cf. OMG's RFP for a Data Distribution Service for Real-Time Systems [15]). A real-

time data flow specification for Java is outside the current scope of the DRTSJ.

The message passing provided by typical OS's used in real-time and distributed real-time systems forces any end-to-end timeliness (and integrity) to be a higher level responsibility. Most frequently, distributed real-time becomes the responsibility of each of the application programmers; less frequently, custom distributed real-time middleware is created. As commercial distributed real-time infrastructure products (notably real-time CORBA-compliant ones) continue to emerge and evolve, they are becoming more widely used.

Java already includes a model for distributed systems - Remote Method Invocation (RMI) [23] - and JSR-50 proposed to enhance it for real-time systems. RMI provides the familiar distributed object system control flow model of method invocations using abstract interfaces to define local stubs for remote objects. As in any distributed object system, programming abstraction similar to asynchronous message passing can be provided by allowing asynchronous (one-way with no return parameters) method invocations. RMI also can transfer object instances by value. This allows messages to be passed as objects. It also supports a simple form of data flow for point-to-point flow of modestly sized data, but not for effective publish/subscribe models.

But RMI by itself intentionally provides very limited support for end-to-end properties and, in particular, was not intended for real-time systems. Therefore, the DRTSJ has the objective of integrating RMI with the RTSJ to support end-to-end timeliness by creating a control flow model that is both a programming model itself and a possible mechanism for other distributed system models such as point-to-point data flow, messaging, and Java's mobile objects.

A distributed object system (as opposed to, for instance, web services) requires a well-defined architecture with stable interfaces between components and some level of system-wide agreement on infrastructure technology. In a real-time distributed object system that agreement includes the semantics of timeliness and sufficiently synchronized local clocks. Typically, such systems are found inside an enterprise - it is difficult to create the technical agreement and coordination needed to build a distributed object system between enterprises. This is consistent with the current normal deployment of distributed real-time computing systems. The scalability requirement for the DRTSJ is based on this presumption of intra-enterprise environments.

3 RMI and Java

This paper assumes that the reader is familiar with the RTSJ and RMI. However, in order to integrate the two, it is necessary to interpret some of the key design goals of RMI. For our purposes, the main components of RMI can be considered as follows:

- the programming model, where objects that can be accessed remotely are identified via a remote interface,
- the implementation model, which provides the transport mechanisms whereby one Java platform can talk to another in order to request access to its objects, and
- the development tools (e.g., rmic or its dynamic counterpart), which make server objects and generate the proxies required to facilitate the communication.

Key to developing RMI-based systems is defining the interfaces to remote objects. RMI requires that all objects that are to provide remote interfaces must indicate so by extending the pre-defined interface `Remote`. Each method defined in an interface extending `Remote` 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 [12]. The location of the remote objects may be transparent, but the fact that remote access may occur is not transparent.

3.1 The meaning of time

In standard Java, time is expressed as a number of milliseconds and nanoseconds past midnight on January 1st 1970. The types of these values are `long` and `int` respectively. 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.

RMI is silent on the issue of the relationship between the clocks on the client and server sites. However, the leasing mechanism of the distributed garbage collection algorithm assumes that the clocks progress approximately at the same rate [19]. If this assumption is violated then remote objects might unexpectedly disappear resulting in an appropriate remote exception being raised when access is attempted.

3.2 Failure semantics

RMI assumes a reliable transport mechanism (e.g. TCP). Consequently, it does not need to be concerned with transient communication errors. RMI's handling of node and permanent communication failures is reentered through the throwing of a `RemoteException`. A `RemoteException` thrown if client make remote call and the RMI implementation is either unable to make the call or makes the call but detects failure before the call has returned.

The server is informed if client node fails while server is executing requests on its behalf. RMI does not implement orphan extermination [8, 16]. Hence, RMI has

exactly-once semantics in the absence of failures and *at most-once* semantics in the presence of failures [18].

4 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 [20]. By implementing an interface, the server is guaranteeing to provide the functionality implied by the interface. *By definition, Java interfaces says nothing about the real-time properties of the server (or in deed any other non-functional requirements). Similarly, re mote interfaces says nothing about the real-time properti es of either the server or the underlying transport syste m.* Consequently, an object that implements a remote Ja va interface is an object that is executing *without* the requirements of a real-time Java platform. Furthermo re, the transport protocols that implement the connection between the client and server are not required to b e timely, irrespective of whether the client is a rea l-time client. If these assumptions are accepted, then the tools that generate the client and server proxies can do so without knowledge of whether the server or the clie nt will execute (or is executing) on a Java platform or a real-time Java platform [1].

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 view ed as an ordinary Java thread. Even if the client is a real-time thread, the proxy thread is viewed as a normal Java thread. This, therefore, is a minimal integration b etween the RTSJ and RMI (called Level 0 integration). Rea l-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 th at it has. Consequently, the application programmer must explicitly pass any scheduling or release parameter s between the client and the server, and require a sympathetic RMI implementation.

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

[1] Of course, this is not the only interpretation of how RMI should be used in a real-time environment. Another one is that the RMI tools and implementation mechanisms should be modified so that they can identify whether the client is a real-time client and whether the server is able to exploit the real-time JVM. This is not the approach propose d in this paper because it requires changes to RMI and is likely to have greater run-time overheads. Moreover, assuming that its ob ased or explicit server attributes that are readily expressed by the client might be more difficult for client to determine the service tha t can be expected.

5 Real-Time RMI (Level 1 Integration)

The discussion presented in section 4 suggests that when a real-time RTSJ thread makes a RMI call an expe ct to timely delivery of the request to the server and th e server will not inherit any of the timing requirements of the client.

In keeping with the non-transparent RMI philosophy, to obtain real-time remote communication Level 1 integration proposes the introduction of a real-tim e RMI. Real-time RMI consists of the following components.

- the programming model, where real-time objects that can be accessed remotely are identified via a real-time remote interface (this is similar to the appro ach that de Miguel has adopted for the addition of a predictable RMI using reservation-based scheduling techniques [10]),
- the implementation model, which provides timely transport mechanisms whereby one RTSJ platform can talk to another in order to request access to its objects and also pass across any timing constraints or scheduling parameters associated with the client [2], and
- the development tools (e.g., a modified rmic), which take server objects and generate the real-time prox ies required to facilitate the real-time communication.

Key to developing real-time RMI-based systems is defining the interfaces to remote real-time objects (objects that assume that they are executing on a RTSJ platf orm). Real-time RMI requires that all objects that provid e a remote interface must indicate so by extending the pre-defined interface `RealtimeRemote`.

```
public interface RealtimeRemote
  extends java.rmi.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 an RTSJ `RealtimeThread` that inherits appropriate scheduling and release parameters from the client. Where the client is not a RTSJ thread but a standa rd Java thread, default release and scheduling parameters c an be provided. Clients that are asynchronous event handl ers need further consideration. Asynchronous event hand lers are expected to be light weight, if they can call re mote objects this may well add to the cost of their implementation. Here it is assumed that they are a llowed and that the proxy thread created at the server is the same as a real-time thread.

[2] Beyond the scope of this paper to address is issues associated with locating real-time servers, issues associated wit h how clients can determine the real-time properties of the servers, its connectio ns and possible that the real-time mechanisms will be base d on other, e.g. JavaSpaces [24] technologies.

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 an RTSJ `NoHeapRealtimeThread`.

One issue that must be considered is whether there needs to be any changes to the Java Serialization [22] mechanisms to support the passing of objects across real-time remote or no-heap real-time remote interface. If so, this can be catered for by producing extensions to the `java.io.ObjectOutputStream` and `java.io.ObjectInputStream` classes.

5.1 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 RTSJ platforms. For example, as far as the server RTSJ platform 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 RTSJ platform 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 time classes are not serializable. This means that any time values passed across the RTSJ RMI must be converted into suitable types and reconstructed at the server site.

5.2 Failure semantics

Level 1 integration assumes that sites suffer only crash failures [6].

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 communication mechanisms. Either an asynchronously interrupted exception (AIE) can be thrown at the server proxy real-time thread, or an asynchronous event (AE) can be fired.

5.3 Limitations 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 two main sets of limitations of the approach.

- limitations from the RMI definition and its interaction with the JVM
- limitations from the RTSJ definition

The limitation from RMI/JVM that is of main concern here is the issue of thread synchronization. In RMI, the following situation will result in deadlock. Client thread calls a synchronized method on Object A, which calls a remote method on Object B. The called method on Object B then directly (or indirectly) calls a synchronized method in Object A. As the JVM and RMI do not collectively support the notion of distributed thread, the proxy thread executing the method on Object A is not considered the same as the original client thread. Note that, since locking is performed on a per-thread basis, there would be no deadlock in the case where Objects A and B reside in a single Java Virtual Machine and use only local, not remote, method invocations.

A similar problem occurs with thread-local data. Here, however, the problem is that data read by the second call to Object A will not be the data that is read by the original call to Object A. In both these cases, the problem is that the proxy thread is not considered to be the same thread as the original client thread.

One way to solve this problem requires the introduction of a global thread identifier that is carried with all RMI calls. The JVM (and RTJVM) needs to be changed in order for it to check this identifier. For the recursive synchronized method situation described above, the RTJVM can either allow the thread to enter the method, or raise a remote exception indicating that deadlock has occurred. Allowing the proxy thread to enter, whilst conceptually simple, has repercussions for the failure semantics [27]. Essentially, under certain failure conditions a naïve implementation will result in two threads being active in the same synchronized object. Raising an exception is simple to implement. The global thread identifier allows thread-local data to work correctly.

The limitation imposed by the RTSJ is more difficult to circumvent. The problem stems from the fact that none of the RTSJ class definitions define 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.

6 Distributed Real-Time Threads (Level 2 Integration)

The full distributed real-time thread model, as discussed in section 2, 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 called the **head** site of the thread. The site on which the distributed thread object was created is called the **root** or **origin** site. Other sites that are hosting part of the distributed thread are called **segment** sites. 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 threads is to extend the RTSJ's `RealtimeThread` class and to implement real-time remote interfaces which define the remote operations that can be called on the thread. However, to implement distributed threads requires more from the underlying RTSJ Java platform and real-time RMI transport protocols than is implied by the discussion in section 5. Consequently, to indicate this, the notions of a distributed real-time JVM (DRTJVM) and distributed real-time RMI are introduced. Distributed real-time RMI consists of the following components:

- the programming model, where distributed real-time objects that can be accessed remotely are identified via a `DistributedRealtimeRemote` interface;
- the implementation model, which provides timely transport mechanisms whereby one DRTSJ platform can make another request access to objects and also to pass scheduling parameters and state information associated with the client in order to facilitate implementation of the distributed thread model;
- the development tools (e.g., a modified real-time rmic), which take server objects and generate the real-time proxies required to facilitate the real-time communication; the server proxy thread can be considered to be a distributed real-time thread.

The distributed real-time Java platform is a real-time Java platform augmented with facilities to support the distributed real-time thread model.

6.1 The Distributed Real-Time Specification

Although it is beyond the scope of this paper to define completely a distributed real-time specification for Java (DRTSJ), the following discusses the main components associated with the distributed thread model.

Distributed Real-Time Threads

The operations that can be performed on a distributed real-time thread 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. One of the motivations for providing these remote operations comes from an assumption that threads on one node in the distributed system might need to perform a mode change on threads whose origin are on other nodes. In order to achieve this, they need to be able to manipulate the threads' release and scheduling parameters. (Of course, the same functionality could be achieved by having the application define its own objects local to the threads' origins and for those objects to perform the mode change. However, allowing direct remote access offers more flexibility. For example, it allows the underlying DRTSJ platform to optimize finding the location of the head of the thread in order to carry out the operation. It also allows a level of fault tolerance when the origin of the thread is not accessible due to node or communication failures.)
- *Operations that affect the execution state of the distributed thread.* These include being able to start a remote thread and to interrupt it. Whilst there is a clear requirement for one node to start a thread on another node, it can be argued that being able to interrupt a thread should not be a remote operation. This topic is deferred until the AIE discussion below.

Figure 1 illustrates the remote operations that might be defined along with the definition of a distributed real-time thread class. The RTSJ classes definitions for `ReleaseParameters`, `SchedulingParameters`, etc. will need to be either re-defined (or subclasses created) to implement remote interfaces or new classes created. (Note that, the reference semantics of Java means that in RTSJ if, for example, the `ReleaseParameters` of a real-time thread are changed then the scheduler implements the changes "immediately". However, in RMI, if the parameters are serializable then a copy is made when they are passed across the network. Consequently the above interfaces shows the parameters as remote interfaces to ensure consistent semantics with the RTSJ.) A new static method allows the current remote thread unique identifier to be found. A similar definition can be derived for `DistributedNoHeapRealtimeThread`.

Asynchronous Events

As well as supporting real-time threads, the RTSJ also supports the notion of events and event handlers. Event handlers may be permanently bound to threads so bound to thread at run-time. The RTSJ is not clear in the case about how many handlers are bound to each schedulable thread.

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

  RemoteScheduler getScheduler() throws RemoteException;

  synchronized void  interrupt() throws RemoteException;

  void  start() throws RemoteException, IllegalThreadStateException;
}

public DistributedRealtimeThread extends RealtimeThread implements RemoteThread
{
  // Implementation of RemoteThread interface plus
  public static  RemoteThread currentRemoteThread()
                 throws  NotARemoteThreadException;
}
```

Figure 4: Distributed Real-time Threads

There are several issues to consider when adapting this model for execution in a distributed environment. It is assumed that the same restrictions that were applied to distributed real-time threads are to be applied to events and event handlers, namely that they are not serializable and cannot be made persistent.[2] Both events and event handlers can have remote interfaces, and the firing of an event on one node can result in the scheduling of a handler on another. A distributed handler is bound to a distributed real-time thread and hence can make remote calls.

One possible definition of the interfaces and classes needed for distributed events and their handlers is given in Figure 2. These definitions allow the remote firing of an event and remote handlers to be attached and manipulated. The `bindTo` method is the allowable remote operation. The only available operation in the `RemoteAsynchronousEventHandler` interface is to get the remote scheduler. The remote node uses this in order to determine which schedulers should be informed when an event has been fired. There is no need for a remote `Runnable` to be made visible. The RTSJ class hierarchy for event handlers is mirrored for global event handlers.

Asynchronous Transfer of Control

The facilities for handling asynchronous transfers of control (ATC) in any language are intrinsically complex. The RTSJ facilities are based on the raising and handling of asynchronous exceptions (using the `AsynchronouslyInterruptedException` (AIE) class and the `Interruptible` interface). In order to be backward compatible with standard Java, the approach is integrated with the interrupt mechanism. Although ATCs are intrinsically complex, there is a strong user requirement for them [3]. These requirements stem from the underlying need for a thread to be able to respond to asynchronous event quickly and safely. This underlying need is independent of whether the thread is a distributed real-time thread or a real-time thread.

Unfortunately, applying ATCs to a distributed thread adds to the inherent complexity of the single processor model. This is because the AIE can be fired on any node in the network that has access to the thread or to the AIE object being executed by the thread (or must be directed to the (moving) head of the thread). Furthermore, there may already be outstanding AIEs for the thread. Careful consideration needs to be given to an implementation model that will support the current RTSJ semantics efficiently. Either,

1. all outstanding AIEs will need to be stored on the thread's origin and every time a thread becomes interruptible it has to check with the origin to see if an AIE should be thrown; this would seem to be prohibitively expensive unless further restrictions are made, for example, AIEs are deferred while a remote call is in progress. Furthermore, the approach would be fragile in the event of failures. Or,

[2] If events were serializable, problems would occur with the event that was bound to an interrupt on another node because its occurrence would have to be passed across the network. Handler could be made serializable, however the model would become confusing with the event that was unclear when a handler is being scheduled.

```
public interface RemoteAsynchronousEvent extends
         DistributedRealtimeRemote
{
   void addRemoteHandler( RemoteAsynchronousEventHandler handler)
                        throws RemoteException;
   // also handledBy, removeHandler, setHandler
   // createReleaseParameters for remote handlers

   void fire() throws RemoteException;
}
public class GlobalAsyncEvent extends AsyncEvent
             implements RemoteAsynchronousEvent { … };

public interface RemoteAsynchronousEventHandler
                 extends DistributedRealtimeRemote
{
  RemoteScheduler getScheduler() throws RemoteException;
}
public class GlobalAsyncEventHandler extends AsyncEventHandler
                implements RemoteAsynchronousEventHandler;

public class GlobalBoundedAsyncEventHandler
                extends GlobalAsyncEventHandler;
```

Figure 2: Asynchronous Events

2. about outstanding AIEs have been carried with the thread of the thread [4].

In the latter case, finding the thread might be difficult but at least the expense is only incurred when the facility is used.

Assuming that AIEs for distributed threads are required, it is necessary to decide the remote operations that can be performed on them. The operations on an AIE can be partitioned into two sets: those required by the interrupting thread and those required by the thread to be interrupted. Currently, the RTSJ does not make this distinction clear; consequently a thread firing an AIE can also call the methods to enable/disable the AIE and, more strangely, the `happened` method [5].

One possibility is to define two remote interfaces. The first is for the thread that wishes to fire the AIE. The second interface is for the use of thread that will be interrupted. These interfaces along with the other associated classes are given in Figure 3. Here a remote thread is allowed to execute the `doInterruptible` method — a thread can, therefore, only execute local `doInterruptible`. The `run` method, however, can make remote calls and consequently the parameter to `run` must be changed to take an interface parameter. A restriction that the `interruptAction` should be local allows the handling routine to access any parameter that might have been declared with the AIE.

Given that `AsynchronouslyInterruptedException` implements `Serializable`, consideration needs to be given as whether it is appropriate for a distributed asynchronously interrupted exception to be passed across the network. If this is allowed, what are the semantics if `doInterruptible` is in progress?

Having `interrupt` in the remote interface of a DistributedRealtimeThread allows a remote thread to have the generic AIE thrown.

Exporting Distributed Real-Time Objects

In Java, references to remote objects can be passed across the network if the server object has been exported. This can be achieved either by defining the remote object to inherit from one of the sub classes of `java.rmi.DistributedRealtimeRemoteObject` or by calling the static method `exportObject` with the remote object as a parameter. In this paper it has been assumed that all relevant classes will have constructors that call `exportObject`.

[4] Other thread control information may need to be passed with the thread for example, security information, whether the thread is a daemon, etc.

[5] The same argument could be made for asynchronous events and one possibility is also a remote fire interface. This would allow a thread to fire an asynchronous notification without being concerned as to whether the result is an AIE or the scheduling of a handler.

```
public interface RemoteFirer extends DistributedRealtimeRemote
{
  boolean fire() throws RemoteException;
}

public interface RemoteAsynchronouslyInterruptedException
   extends DistributedRealtimeRemote
{
  boolean disable() throws RemoteException;
  boolean enable() throws RemoteException;
  boolean isEnabled() throws RemoteException;
  boolean happened(boolean propagate) throws RemoteException;
}

public class DistributedAsynchronouslyInterruptedException
    extends AsynchronouslyInterruptedException
    implements RemoteFire, RemoteAsynchronouslyInterruptedException
{
    // implement the interfaces and
   public boolean doInterruptible(DistributedInterruptible logic);
}
public interface DistributedInterruptible
{
  void  interruptAction (AsynchronouslyInterruptedException exception);
  void  run (RemoteAsynchronouslyInterruptedException exception)
        throws AsynchronouslyInterruptedException;
}
```

Figure 3: Asynchronous Transfers of Control

6.2 The meaning of time

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

6.3 Failure semantics

Sites are assumed to suffer from only crash failures. In the distributed thread model, the DRTSJ platform is directly supporting the distributed thread semantics. Consequently, when a segment site hosting the distributed thread fails, the DRTSJ platforms 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. Level 1 integration (i.e., ignore, throw an AIE, fire an AEH). 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, which point the remote exception is lost.

When the thread handling the remote exception or AIE or running the AEH handler, details of the failure can be found from the underlying DRTSJ platform.

7 Interoperability

The following table shows the level of interoperability between the various models. Note that for all cases other than a Java thread, remote calls may be subject to ATC. Consequently, this issue is given special consideration in the table below. The table also considers only remote, real-time remote and distributed real-time remote interfaces. The no-heap variations introduce new problems not other than the requirement for heap usage.

Client Class	Remote Interface	Real-time Remote	Distributed Real-time Remote
Thread	Defined by RMI Server proxy thread is viewed as standard Java thread No AIE Expressible from client	Default timing properties used by real-time RMI Server proxy thread is viewed as real-time Java thread with default scheduling parameters No AIE Expressible from client	Default timing properties used by distributed real-time RMI Server proxy thread is viewed as new distributed real-time Java thread with default scheduling parameters No AIE Expressible from client
Real-Time Thread or (Bound) Asynchronous Event Handler	Defined by RMI Server proxy thread is viewed as standard Java thread No AIEs are propagated through RMI as remote method throws an AIE and there is no distributed thread model	Client timing properties are used by real-time RMI Server proxy thread is viewed as real-time Java thread with inherited scheduling parameters and sporadic release parameters AIEs can be declared on remote method, however, as no distributed thread model, only the client thread can be interrupted; this is propagated through the real-time RMI system to the proxy thread	Client timing properties are used by the distributed real-time RMI Server proxy thread is viewed as new distributed real-time Java thread with appropriate release and scheduling parameters AIEs can be declared on remote method, the client real-time thread is interrupted is propagated the server machine if the distributed thread is interrupted this is propagated by the DRTSJ platform(s) to the head of the distributed thread
Distributed Real-Time Thread or (Bound) Global Event Handler	Defined by RMI Server proxy thread is viewed as standard Java thread No AIEs are propagated through RMI as remote method throws an AIE and there is no distributed thread model	Client timing properties are used by real-time RMI Server proxy thread is viewed as real-time Java thread with inherited scheduling parameters AIEs can be declared on remote method, however, as no distributed thread model only the client distributed thread can be interrupted; this is propagated through the real-time RMI system to the proxy thread	Client timing properties are used by the distributed real-time RMI Server proxy thread is viewed as segment of the distributed real-time Java thread AIEs can be declared on remote method, the distributed thread can be interrupted from any site that has access the distributed real-time object can any associated AIE object; this is propagated by the DRTSJ platform(s) to the head of the distributed thread

8 Conclusions and Future Work

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 of 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 RTSJ or the RT JVM.
- Real-time Java threads can call real-time remote objects and they can expect timely delivery of messages, inheritance of scheduling parameters and an

associated global thread identifier. However, they have no other distributed thread functionality — there are extensions required to RMI but no extensions required to RTSJ but a small extension to the supporting platform is required.

- 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 the supporting platform.

Currently, the authors are considering the details of the Real-time RMI infrastructure and the approaches to support distributed real-time scheduling. It is expected that the latter will follow closely the approach adopted by dynamic real-time CORBA [14].

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. M. Boger, *Java in Distributed Systems*, Wiley, 2001.
2. G. Bollella et al, *The Real-Time Specification for Java*, Addison Wesley, 2000.
3. A. Burns and A.J. Wellings, *Real-Time Systems and Programming Languages* 3rd Ed, Addison Wesley, 2001.
4. 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 the Kernel Architectures, pp 200-208, 1993.
5. R. Clark *et al*, *An Adaptive, Distributed Airborne Tracking System*, Proc. Workshop Parallel and Distributed Real-Time Systems, IEEE, http://www.real-time.org/docs/wpdrts99.pdf, 1999.
6. F. Cristian, Understanding Fault-Tolerant Distributed Systems, CACM, 34(2), pp 56-78, Feb 1991.
7. E. Jensen, *The Distributed Real-Time Specification for Java: An Initial Proposal* the Journal of Computer Systems Science and Engineering, March 2001.
8. B. Liskov *et al*, *Orphan Detection*, IEEE 17th Fault-Tolerant Computing Symposium, pp 2-7, July 1987.
9. D. Maynard *et al*, *An Example Real-Time Command, Control, and Battle Management Application for Alpha*, Archons Project TR-88121, CMU Computer Science Dept., http://www.real-time.org/docs/c2demo.pdf, December 1988.
10. M. Ade Miguel, *Solutions to Make Java-RMI Time Predictable*, The Fourth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, pp 379-386, 2001.
11. J.D. Northcutt, Mechanisms for Reliable Distributed Real-Time Operating Systems: The Alpha Kernel, Academic Press, 1987.
12. R. Oberg, *Mastering RMI*, Wiley, 2001.
13. OMG, *Enhanced View of Time, Revised Submission*, OMG Document orbos/99-10-02, October 1999.
14. OMG, Dynamic Scheduling Real-Time CORBA 2.0, Joint Final Submission, OMG Document orbos/2001-04-01.
15. OMG, Data Distribution Service for Real-Time Systems, Request for Proposals, ORBOS/2001-09-11, Object Management Group, September 2001.
16. F. Panzieri and S.K. Shrivastava, Rajdoot: A Remote Procedure Call Mechanism Supporting Orphan Detection and Killing, IEEE TOSE, 14(1), pp 30-37, 1988.
17. M. Philippsen and M. Zenger, *JavaParty – Transparent Remote Objects in Java*, Concurrency Practice and Experience, 9(11), pp 1125-1242, 1997.
18. E. Pitt and K. McNiff, *The Remote Method Invocation Guide*, Addison Wesley, 2001.
19. E. Pitt, *Personal Communication*, October 2001.
20. Recursion Software, *Voyager*, http://www.recursionsw.com/products/voyager/, accessed February 2002.
21. J. Rumbaugh, I. Jacobson and G. Booch, *The Unified Modeling Language Reference Manual*, Addison Wesley, 1999.
22. Sun Microsystems, Java Object Serialization Specification, December 1998.
23. Sun Microsystems, Java Remote Method Invocation Specification, December 1999.
24. Sun Microsystems, *JavaSpaces Service Specification*, Version 1.1, October 2000.
25. Sun Microsystems, *Jini Architecture Specification*, Version 1.1, October 2000.
26. D. Wells, *A Trusted, Scalable, Real-Time Operating System*, Dual-Use Technologies and Applications Conference Proceedings, pp 262-270, 1994.
27. A. Wollrath, *Personal Communication*, October 2001.