The Distributed Real-Time Specification for Java: A Status Report

Ray Clark
The MITRE Corporation

E. Douglas Jensen
The MITRE Corporation

Andy Wellings
Department of Computer Science, University of York

Doug Wells
The Open Group

Abstract

This paper summarizes the status as of December 2001 of the Distributed Real-Time Specification for Java (DRTSJ), being developed by the JSR-50 Expert Group in Sun's Java Community Process. The paper first defines what "real-time," "distributed," and "distributed real-time" mean in the context of the DRTSJ, since those terms are commonly used in widely different ways. The DRTSJ is focused on supporting predictability of end-to-end timeliness for sequentially trans-node behaviors (e.g., chains of invocations) in dynamic distributed object systems. The DRTSJ introduces the Distributed Real-Time Remote Method Invocation (RMI) model, based on the Alpha distributed real-time OS kernel's distributed threads model. The OMG Dynamic Scheduling Real-Time CORBA 2.0 specification also employs a distributed threads model, so these two distributed real-time computing system programming model standards are relatively congruent. The DRTSJ consists of extensions to the Real-Time Specification for Java (RTSJ), Java's RMI, and the RTSJ Java Virtual Machine. The DRTSJ, including the reference implementation, is expected to be released by the end of calendar 2002.

1 Introduction

The Real-Time Specification for Java (RTSJ) [RTSJ 01] specifies what is likely to be the first real-time programming language that is both technically and commercially successful.

Distributed real-time computing systems offer many advantages and opportunities to the real-time computing domain, just as networking and distributed systems do to the non-real-time computing domain. The JSR-1 Expert Group [http://www.rtj.org] deliberately deferred consideration of distributed real-time systems when writing the RTSJ. Sun subsequently formed, and designated Jensen to lead, the JSR-50 [JSR-50] Expert Group [http://www.drtsj.org] to write the (first) Distributed Real-Time Specification for Java (DRTSJ).

The initial proposal for the DRTSJ was described earlier [Jensen 01]. The primary status report for the DRTSJ as of the writing of this paper in December 2001 appears in Section 3 – a framework for integrating the RTSJ and Java's Remote Method Invocation (RMI) [Wellings et al. 01]. 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. The framework also allows interoperability between a standard Java Virtual Machine (JVM), a real-time JVM, and a distributed real-time JVM. The framework as described here includes three (others are being considered) distributed real-time models, in an increasingly functional (and complex) hierarchy of extensions. One (or perhaps a modification of one) of these models will be selected for the DRTSJ.

First, it is necessary to identify what "real-time," "distributed," and "distributed real-time" mean in the context of the DRTSJ. All three terms are frequently used with widely different, usually imprecise, meanings. Without a common lexicon, it would be very difficult for the JSR-50 specification writers to write the DRTSJ, and for prospective users to assess it with respect to their requirements. Section 2 summarizes how these three terms are used in the DRTSJ.

2 "Real-Time" and "Distributed Real-Time" in the DRTSJ

There are numerous widely different products, applications, and systems called either "real-time" or "distributed real-time." In this section, we briefly identify what those terms mean in the context of the DRTSJ.

2.1 Real-Time

The real-time practitioner (user, vendor) community uses the term "real-time" in many different, imprecise, ways. The real-time research community essentially has consensus on a precise definition for "hard real-time" (different from the various things most of the practitioners usually mean), but not on one for "soft real-time" (other than the tautological "not hard real-time"). A broad range of people speak of "real-time" in the sense that "on-line" has historically been used (e.g., "real-time financial information"). It is important for the successful creation and use of the DRTSJ that its writers and users share a crisp understanding of some fundamental concepts of real-time computing.

The need for more precise and general real-time terminology was recognized by the NIST-sponsored workshops that developed the initial requirements for "real-time Java" [Carnahan and Ruark 99]. That group adopted a relatively rigorous real-time lexicon (provided by Jensen), which appears in "Section 1 Concepts" of their document. It is briefly recapped and extended here.

Informally and at a high level of abstraction, an application is a *real-time* one to the degree that timeliness is an intrinsic part of the logic of the application. Timeliness is about satisfying application time constraints acceptably well with acceptable predictability. An application or system *operates in real-time* to the degree that the applications' timeliness requirements are satisfied. An application or system operates in real-time (to whatever degree) due to some combination of: off-line (by human or software tool) or on-line (by OS, middleware, etc.) resource management that explicitly takes application timeliness requirements into account to some extent; resource over-capacity.

Operating in real-time is not about being "real fast;" time-frames (of time constraints, execution durations, etc.) are application-specific and may range from microseconds to megaseconds. A computing system is a real-time one to the degree that it employs resource management that explicitly takes application timeliness requirements into account; that entire spectrum of systems is called *real-time computing systems*.

At a greater level of detail, timeliness involves two dimensions – how well application time

constraints are satisfied, and how predictable that satisfaction is.

2.1.1 Optimality of Timeliness

There are two aspects of the dimension regarding how well application time constraints are satisfied.

First, it must be possible for application designers or users to express a time constraint for the execution of an "action" – e.g., a thread executing some delineated time-critical region of code. A deadline is a familiar, although very simple special case of, a time constraint; *time/utility functions* are a more general formulation [Gouda et al. 77, Jensen 87, Jensen 00].

Second, a system generally executes a multiplicity of both time-constrained and non-time-constrained actions, with real or virtual concurrency. Thus it must be possible for application designers or users to express any of a variety of application- or even situation-specific policies for managing resources – so as to satisfy the actions' time constraints or other execution eligibility criteria acceptably well, while resolving dependencies, especially contention for shared resources, among the actions.

The most visibly managed resource in computer systems is a processor. Managing it consists of sequencing – scheduling, or in the absence of scheduling, dispatching – execution entities (say, threads). Coherent sequencing policies should be used to manage as many as possible of the system's other physical and logical resources (e.g., network, synchronizers) along with the processor(s), to reduce the number or duration of action timeliness anomalies.

2.1.2 Predictability of Timeliness Optimality

The second dimension of timeliness is the distinguishing performance metric of real-time computing: how predictable the first dimension is. It is described here as the second dimension because it is easier to explain after the first dimension is explained.

"Predictability" is virtually never defined by either practitioners or researchers in the real-time computing community. A primary reason is that traditional real-time computing theory and practice are historically focused mostly on simple, static, device-level subsystems. There it is usually presumed that the first dimension of timeliness is limited to only always meeting all deadlines, and that the second dimension (predictability of the first dimension) is limited to only knowing deterministically whether or not all the deadlines will always be met – i.e., the classical hard real-time case. In general, and especially in more dynamic cases, such as exist above the device level in most distributed real-time systems, the hard real-time model is at least inefficient and often counter-productive. Thus there is a need for being able to make more general predictions about the first dimension of timeliness.

Informally, something is predictable to the degree that it can be known in advance. For example, with deadline time constraints, a system designer or user might want some application-specific degree of predictability of (in order of increasing detail and difficulty to predict): whether or not all deadlines will be met; how many deadlines will be met; what the mean (or maximum, etc.) tardiness will be; what the time-constrained action completion times will be. With more general time constraints, such as time/utility functions, properties of accrued utility need to be predicted – e.g., bounds (minimum, maximum, etc.), central tendencies (mean, mode, median, etc.),

dispersion (variance, coefficient of variation, kurtosis, etc.).

Making a prediction means taking information with various kinds and degrees of uncertainties, and then logically deducing or inferring consequential conclusions having various kinds and degrees of uncertainties. In computing (and various other) systems, these conclusions are normally used for resource management to achieve the desired behavior. In computing systems, the information needed for predictions is about the actions' properties, resource requirements, and dependencies, the system behavior, the characteristics of the system's execution environment, etc.

There is a second, implicit, reason why "predictability" is rarely defined at all, and never precisely, in the real-time computing community (among others). Although predictability seems an intuitive notion, it is a surprisingly elusive and complex concept in principle and practice.

The non-obvious nature of predictability is easily glimpsed by a simple example. Consider probability density functions (pdf's). The deterministic distribution ($p = 1$ at $X = k$) is obviously the most predictable one. It might seem intuitive that the least predictable distribution would be the uniform distribution ($p = P$ for all X). But an alternative intuitive perspective is that predictability of a pdf is inversely proportional to its variability – as measured, for example, by its coefficient of variation $C_v = \text{variance}/\text{mean}^2$. From that perspective, the deterministic distribution, whose $C_v = 0$, is still the most predictable; but the standard form of the uniform distribution has a relatively low $C_v = 0.58$ and thus relatively high predictability, and many distributions exist that have higher C_v's and are thus less predictable – for example, the extreme mixture of exponentials distribution has an arbitrarily large C_v and is thus arbitrarily unpredictable.

2.2 Distributed and Real-Time Distributed Systems

For the purpose of the DRTSJ, the term *distributed system* informally refers to a computing system whose programming model is based on there being application-level entities that exhibit multi-node behaviors.

Multi-node behaviors have multi-node properties that must be maintained. The primary multi-node property of interest in real-time distributed computing systems is end-to-end timeliness, which will be discussed in the following section. A critical multi-node property of almost all distributed systems is the integrity of the multi-node behaviors in the face of partial (node, network) failures – e.g., orphan detection and elimination; that will be touched on in Section 2.2.2. Certain types of distributed systems also have multi-node properties specific to them – e.g., control of remote execution points – as will be mentioned when the DRTSJ's programming model is introduced in Section 2.2.3.

2.2.1 End-to-End Timeliness

In real-time distributed computing systems, multi-node application behaviours may have end-to-end time constraints. The defining characteristic of any real-time distributed computing system, whatever its programming model, is that the timeliness (optimality and predictability of optimality) of each multi-node application behaviour – on each individual node it involves, and collectively end-to-end on all nodes it currently involves – is acceptable to that application under the current circumstances.

The "current circumstances" include the latency characteristics of the underlying infrastructure (node OS's, network), the attributes of the

system load, etc. All those circumstances are known and controlled á priori in static distributed real-time computing systems, notably including hard real-time ones. But in general, not all of those circumstances are (or can be) known and controlled á priori – the system is *dynamic* to the degree that is the case.

The RTSJ is oriented toward dynamic (non-static) systems, although it has been proven to successfully support static hard real-time computing. Similarly, the DRTSJ is oriented toward the general case of dynamic distributed real-time computing systems, although we expect it to also be capable of supporting the special case of static hard real-time ones.

In most cases, the extent to which end-to-end multi-node timeliness can be achieved depends on the extent to which each multi-node application behavior's timeliness properties – e.g., end-to-end time constraints, expected execution time, execution time received thus far, etc. – are explicitly employed for appropriately coherent resource management (particularly sequencing) on each node currently involved in that application multi-node behavior. That includes sequencing of the network connecting those nodes. Thus (in dynamic distributed real-time computing systems) these properties need to be propagated among corresponding computing node resource managers in operating systems, JVM's, middleware, and even (directly or indirectly) device – e.g., network, storage – controllers. Acquiring, propagating, depositing, and employing this information should be done transparently to the application programmers; indeed, often some of this information is not even accessible by the application programmers.

It is well known that in general, global optimality of multi-node sequencing, and hence of multi-node behavior end-to end timeliness, is not feasible – but approaches for acceptably approximating it at affordable costs can be roughly placed into three categories. In the first category, every node currently involved in one or more multi-node behaviors uses a consistent (typically the same) sequencing policy. The Alpha distributed real-time OS kernel [Northcutt 87, Maynard et al. 88, Clark et al. 92], and its direct descendents the MK7 distributed real-time OS system [Wells 94, Clark et al. 98, Open Group 98], IBM's never-released Workplace OS for PowerPC, and Real-Time CORBA 1 (fixed priority) [OMG 99] and 2 (dynamic scheduling) [OMG 01a], are primarily in this category. In the second category, a logically singular global resource sequencer is instantiated on every node, and the instantiations interact to perform sequencing for all the nodes. This can provide improved timeliness over the first category, but potentially at higher costs (e.g., inter-node communication latencies). Alpha, MK7, and Real-Time CORBA are in this category for sequencing of their synchronizers. In the third category, one or more levels of "meta" resource sequencers operate above the node resource sequencers – e.g., as a compromise between the first and second categories, or when it is not feasible to have node resource sequencers in categories one or two. There are numerous examples of such systems, principally for the second reason.

2.2.2 End-to-End Integrity

Almost all distributed systems are (or should be) designed to support maintaining the integrity of their multi-node application behaviors despite partial failures of nodes and the network. For example, in the context of multi-hop chains of RPC's or method invocations, failures usually create orphans that must be cleaned up (execution terminated, resources reclaimed, etc.). In a real-time distributed system, this integrity maintenance usually is subject to time constraints.

2.2.3 Distributed Real-Time in the DRTSJ

There are many different kinds of distributed system models, and the Internet in particular is inspiring more. To focus the DRTSJ on the real-time domain's current mainstream, we triaged the vast and ill-defined field of distributed systems in three steps.

The first triage step was based on the observation that the vast majority of deployed distributed real-time computing systems employ at least one of the following programming models:

- *control flow:* movement of an execution point, with or without data (parameters), among application entities (procedures, objects) – e.g., RPC and remote method invocation
- *data flow:* movement of data, without an execution point, among application entities – e.g., publish/subscribe, and bulk data transfers
- *Networked:* asynchronous (send/no-wait) or synchronous (send/wait) 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. To the extent that the DRTSJ does not support these models effectively, we relegate them to a prospective subsequent distributed real-time Java Specification Request (JSR).

The second triage step took into account the relative suitability of those three programming models for distributed real-time systems as characterized in Section 2.2.

Control flow models are usually designed for multi-node – usually *trans-node* (linearly sequential) – behaviors that are synchronous in the send/wait sense, but they are not limited to that. When a synchronous response is not needed at the programming model level (synchronous returns transparent to the programming model are sometimes used for reliability within the communication subsystem), a one-way invocation can be spawned asynchronously (as in CORBA). The distributed system infrastructure may also support asynchronous "delayed returns" by providing a means for associating subsequent (usually one-way) "reply" invocations with their corresponding one-way "send" invocations.

Precedents exist for a control flow model that shares the trans-node behavioral state needed to support end-to-end timeliness in dynamic distributed real-time computing systems. The first one of that type was 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 – and hence is neither synchronous nor asynchronous. A distributed thread transparently propagates its pertinent properties – notably here, timeliness, but perhaps also resource ownership, transactional context, security attributes, etc. – when its execution point transits object (and perhaps node) boundaries. We will call a distributed thread that propagates timeliness properties a *distributed real-time thread*.

The distributed thread model appeared first in the Alpha distributed real-time OS kernel and was subsequently incorporated in the kernels of the MK7.3 distributed real-time OS and the Workplace OS for PowerPC (among others); it is the basis of the programming model for OMG's recent Real-Time CORBA 2.0 (Dynamic Scheduling) specification. The Alpha and MK7.3 instances of the model also provided support for end-to-end integrity of distributed threads, plus other features such as execution point control and distributed events. Realistic

non-trivial experimental distributed real-time computing systems were successfully constructed using Alpha [Maynard et al. 88] and MK7 [Clark et al. 98].

Almost all data flow models, including the extant "real-time" publish/subscribe ones, 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 [OMG 01b]). Thus, a real-time data flow specification for Java is left to a prospective subsequent JSR.

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.

It would not be sensible for the DRTSJ to be based on a message passing model – instead, it should enable commercial Java platform products for constructing distributed real-time computing systems.

The third triage step was simply to observe that Java already includes a model for distributed systems – Remote Method Invocation (RMI) [Sun 99] – and JSR-50 proposed to enhance that for real-time systems. RMI provides the familiar distributed object system control flow model of method invocations (and normally) returns, using abstract interfaces to define local stubs for remote objects. As in any distributed object system, a programming abstraction similar to asynchronous message passing can be provided by spawning new one-way (no-response) flows. 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 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. Java's RMI does not lend itself to this integration directly. Our approach to accomplishing it is summarized in Section 3.

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 A Framework for Integrating the RTSJ and RMI

3.1 RMI

Key to developing an RMI-based system is defining the interfaces to remote objects. RMI

requires that all objects that are to be accessed remotely provide a remote interface by extending the pre-defined interface, `Remote`. Each method defined in this interface must declare that it "`throws RemoteException`". Thus one of the fundamental 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.

3.2 The meaning of time

In standard Java, time is expressed as a number of milliseconds and nanoseconds since January 1, 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.

3.3 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.

3.4 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. 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 considered to be 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.

3.5 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 a 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.

3.5.1 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 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 real-time RMI must be converted into suitable types and reconstructed at the server site.

3.5.2 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.

3.5.3 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 real-time JVM 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

`AsynchronouslyInterrup-tedException` (AIE), 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.

3.6 Distributed Real-Time Threads (Level 2 Integration)

As described in Section 2.2.3, a distributed thread is one whose locus of control can move freely across the distributed system by calling methods in remote objects. Each distributed 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. The current execution point of the thread is called its *head*. 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. Hereafter, the terms "distributed thread" and "distributed real-time thread" are synonymous.

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 real-time JVM and real-time RMI transport protocols than that implied by the discussion in Section 3.5. 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 `ReleaseParam-eters`, `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.

3.6.1 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).

3.6.2 Failure semantics

Sites are assumed to suffer from crash failures only.

```
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 Real-Time Theads

In the distributed thread model, the distributed real-time JVM is directly supporting the distributed thread semantics. Consequently, when a segment site hosting the distributed thread fails, the distributed real-time JVM's 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 asynchronous event (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.

4 Conclusions

This paper first sought to circumvent the widespread confusion and vagueness about the concepts and terms "real-time," "distributed," and "distributed real-time" by describing what they mean in the context of the DRTSJ. Then it identified a specific class of distributed real-time computing system models that this specification is focused on. Next it explored some of 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 RTSJ or the real-time 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 real-time JVM.

The JSR-50 Expert Group is further considering these and other related options, and will select the model for the DRTSJ.

The DRTSJ, including the reference implementation, is expected to be complete by the end of calendar year 2002.

5 Acknowledgement

The authors gratefully acknowledge the help and advice given by members of the JSR-50 Expert Group.

6 References

Carnahan and Ruark 99 L. Carnahan and M. Ruark (eds.), *Requirements for Real-Time Extensions for the Java Platform*, http://www.nist.gov/itl/div897/ctg/real-time/rt-doc/rtj-final-draft.pdf , National Institute of Standards and Technology, September 1999.

Clark et al. 98 R. Clark, P. Hurley, E. Jensen, T. Lawrence, A. Kanevsky, J. Maurer, P. Wallace, D. Wells, T. Wheeler, and Y. Zhang, *An Adaptive Distributed Airborne Tracking System*, Proc. Workshop on Parallel and Distributed Real-Time Systems, IEEE, http://www.real-time.org/papers/wpdrts_d1.pdf, http://www.real-time.org/presentations/wpdrts99/atd/screens.htm, 1998.

Clark et al. 92 R. Clark, E. Jensen, and F. Reynolds, *An Architectural Overview of the Alpha Real-Time Distributed Kernel*, Proc. USENIX Workshop on Microkernel and Other Kernel Architectures, http://www.real-time.org/papers/usenix92.pdf, USENIX, April 1992.

Gouda et al. 77 M. Gouda, Y. Han, E. Jensen, W. Johnson, and R. Kain, *Distributed Data Processing Technology, Vol. IV, Applications of DDP Technology to BMD: Architectures and Algorithms, Chapter 3, Radar Scheduling; Section 1, The Scheduling Problem*, Honeywell Systems and Research Center, NTIS ADA047475, September 1977.

Jensen 85 E. Jensen, C. Locke, and H. Tokuda, *A Time-Value Driven Scheduling Model for Real-Time Operating Systems*, Proc. Symposium. on Real-Time Systems, IEEE, November 1985.

Jensen 00 E. Jensen, *Utility Functions: A General Scalable Technology for Software Execution Timeliness as a Quality of Service*, Proc. Software Technology Conf., Utah State Univ., April 2000.

Jensen 01 E. Jensen, *The Distributed Real-Time Specification for Java – An Initial Proposal*, the Journal of Computer Systems Science and Engineering, March 2001.

JSR-50 Jensen, E., Java Specification Request 000050, http://java.sun.com/aboutJava/communityprocess/jsr.html, Sun Microsystems, April 1999.

Maynard et al. 88 D. Maynard, S. Shipman, R. Clark, J. Northcutt, R. Kegley, B. Zimmerman, and P. Keleher, *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/papers/gd.pdf, December 1988.

Northcutt 87 J. Northcutt, Mechanisms for Reliable Distributed Real-Time Operating Systems – The Alpha Kernel, Academic Press, ISBN 0-12-521690-4, 1987.

OMG 99 *Real-Time CORBA 1.0 Specification*, ORBOS/99-02-12, and *errata*, ORBOS/99-03-29, Object Management Group, 1999.

OMG 01a *Dynamic Scheduling Real-Time CORBA 2.0 Joint Final Submission*, ORBOS/2001-04-01, Object Management Group, April 2001.

OMG 01b Data Distribution Service for Real-Time Systems, Request for Proposals, ORBOS/2001-09-11, Object Management Group, September 2001.

Open Group 98 *MK7.3a Release Notes*, http://www.real-time.org/papers/RelNotes7.Book.pdf, The Open Group, 1998.

RTSJ 01 G. Bollella et al., *The Real-Time Specification for Java*, V1.0, http://www.rtj.org, December 2001.

Sun 99 Sun Microsystems, *Java Remote Method Invocation Specification*, December 1999.

Wellings et al. 01 A. Wellings, R. Clark, D. Jensen, and D. Wells, *A Framework for Integrating the Real-Time Specification for Java and Java's Remote Method Invocation*, Real-Time Systems Symposium, Work in Progress Session, IEEE, December 2001.

Wells 94 D. Wells, *A Trusted, Scalable, Real-Time Operating System*, Dual-Use Technologies and Applications Conference Proceedings, 1994.