Distributed Real-Time Specification for Java

A Status Report (Digest)

Jonathan S. Anderson
MITRE Corporation
202 Burlington Drive
Bedford, Massachusetts
andersoj@andersoj.org

E. Douglas Jensen
MITRE Corporation
202 Burlington Drive
Bedford, Massachusetts
jensen@real-time.org

ABSTRACT

The Distributed Real-Time Specification for Java (DRTSJ) is under development within Sun's Java Community Process (JCP) as Java Specification Request 50 (JSR-50), lead by the MITRE Corporation. We present the engineering considerations and design decisions settled by the Expert Group, the current and proposed form of the Reference Implementation, and a summary of open issues. In particular, we present an approach to integrating the distributable threads programming model with the Real-Time Specification for Java and discuss the ramifications for composing distributed, real-time systems in Java. The Expert Group plans to release an initial Early Draft Review (EDR) for previewing the distributable threads abstraction in the coming months, which we describe in detail. Along with that EDR, we will make available a demonstration application from Virginia Tech, and a DRTSJ-compatible RTSJ VM from Apogee.

Categories and Subject Descriptors

C.2.4 [**Computer Systems Organization**]: Computer-Communication Networks—*Distributed systems*; C.3.d [**Computer Systems Organization**]: Special-Purpose and Application-Based Systems—*Real-time and embedded systems*

General Terms

Design, Standardization, Reliability

Keywords

Distributed, Real-time, Java, Distributable thread, Thread integrity, Distributed scheduling

1. INTRODUCTION

The Distributed Real-Time Specification for Java (DRTSJ) is under development within Sun's Java Community Process (JCP) by the membership of the JSR-50 Expert Group (EG). This group, led by the MITRE Corporation, includes representation from individual, academic, U.S. government, defense, and industrial participants.[8]

The EG has settled on the following components as a minimum for the final specification:

Distributable Real-Time Threads, a proven programming model for constructing sequential control flow applications with end-to-end timeliness properties in distributed systems. The DRTSJ's distributable threads are a real-time generalization of Java's Remote Method Invocation, as originally proposed in JSR-50 (and are a superset of the abstraction provided in the OMG Real-Time CORBA specification 1.2 [20];

A **Distributable Thread Integrity Framework**, into which application designers may plug appropriate policies for maintaining the health and integrity of distributable threads in the presence of failures; and

A **Scheduling Framework**, into which application designers may plug appropriate user space policies for scheduling distributable and local threads.

In the Fall of 2006, the EG will release an technology preview package called **Early Draft Review #1** (EDR#1) which will focus on the specification and implementation of the distributable threads abstraction. Subsequent Early Draft Reviews may be released to preview the distributable thread integrity framework, and the scheduling framework.

2. GUIDING PRINCIPLES FOR THE DRTSJ

The EG recognizes the enormous variety of potential problem and solution spaces represented by the terms "distributed" and "real-time," and the variety of opinions on what it could mean to construct distributed real-time systems in the Java Programming Language. The JSR-50 proposal and the EG deliberately scoped the DRTSJ to a subset of those solution spaces and approaches to constructing distributed real-time systems, by extending the JSR-1 Real-Time Specification for Java (RTSJ) in a natural and familiar way, especially for Java programmers. We seek in this digest to briefly articulate this scope, summarize the work accomplished to date, and describe the intended products of the JSR-50 specification effort.

Several key assertions were debated and articulated by the EG in order to set the scope and define the relationship of JSR-50 to other work. Many of these deliberately follow the

JTRES '06, October 11-13, 2006 Paris, France

assertions made by the RTSJ EG, upon whose work much of the DRTSJ is predicated. [26, 13, 14] Here we highlight some of these guiding principles, though we include by reference those called out by the RTSJ EG in [3, Introduction].

Bring distributed real-time to Java, not bring Java to traditional distributed real-time. [15, 11] It is the intention of the JSR-50 EG to bring facilities required for constructing the DRTSJ style of distributed real-time systems into the Java programming environment in a way that is least disruptive to experienced Java programmers. For instance, concurrent programming primitives such as the `synchronized` keyword and the `Thread` and related classes should remain familiar. Similarly, we build upon rather than replace Java's distributed object model as expressed in the Java RMI specification. A corollary is that the DRTSJ is, as its name indicates, specifically for Java, and is not intended to be language agnostic (as, for example, CORBA is more or less intended). The opposite approach of bringing Java to distributed real-time could conceivably be performed with a binding of the RTSJ to Real-Time CORBA, requiring Java programmers to learn something about how to use CORBA. Each approach is reasonable for some contexts, and our choice for the DRTSJ is not intended to compete with that alternative (should it eventually materialize), but instead to complement it and provide the users two options (at least).

Distributed objects and operations are distinguished from local objects and operations, for the obvious reasons of latency, partial failures, and concurrency control. [35]

Current Practice vs. Advanced Features: The DRTSJ will address current real-time system practice as well as allow future implementations to include advanced features. The EG has chosen to support this goal by providing a specification which targets those technologies, techniques, and interfaces which have been well-tested, subjected to review, and successfully employed in the construction of non-trivial distributed real-time systems. Additional promising technologies under development by the research community which fail to meet those specification criteria may be provided in versions of the DRTSJ Reference Implementation (RI). Such versions of the RI serve as an active testing ground for innovative technologies which may appear in future versions of the specification.

Maintain the "flavor" of RTSJ: Java programmers who have already made the leap to the RTSJ should find adaptation to the DRTSJ as natural as possible. RTSJ applications should run unmodified on DRTSJ-compliant JVMs, with some caveats on their distributed behavior.

Do not dictate the use of an RTSJVM or real-time transport: The DRTSJ allows (the inevitable) mixtures of regular and RTSJ-compliant JVMs, and specifies the timeliness behaviors that result. It also is deliberately silent on the topic of the transport, which is regarded as a quality of implementation issue. As with Real-Time CORBA, real-time transports can be addressed in a subsequent specification. [21]

Coherent support for end-to-end application properties: By definition, some multi-node behaviors in distributed real-time systems have end-to-end time constraints which must be respected for the system to perform acceptably. Other end-to-end properties may also be required, such as fault management, security credentials, serialization, etc. Traditionally these end-to-end properties have been forced on the application designers who must create bespoke and often ad-hoc mechanisms to attain them, typically at high recurring and non-recurring costs. The DRTSJ must provide basic facilities for passing and acting upon the required end-to-end context among the nodes participating in any given distributed behavior, in a manner that respects both the end-to-end argument [28, 16] and the counter-examples, particularly those found in real-time systems. Conventional message-passing models (e.g., JMS [29]) and publish/subscribe models (e.g., OMGs DDS [22]) tend to disregard the common need for multi-hop interactions, and hence make end-to-end properties the responsibility of the users.

Based on these driving premises, the keystone elements of the DRTSJ outlined in the introduction were selected. The distributable threads model, discussed in detail in Section 3, provides a coherent, end-to-end abstraction for building concurrent, sequential activities for distributed systems in a manner familiar to Java and real-time programmers alike. In particular, the presence of a distributed object model (RMI) in the Java platform makes the distributable thread concept a straightforward step for Java programmers into the world of distributed real-time programming.

In non-trivial distributed systems, partial failures due to changing network conditions, node failures or overloads, and even regular maintenance must be considered as the common case rather than the exception. Therefore, any distributed system must provide facilities for detecting or masking these failures, and presenting the relevant events to the application. Again, the end-to-end argument and its counter-examples must drive engineering solutions in which responsibilities are assigned to different levels of any given system. The concept and implementations of distributable thread integrity, described in Section 3.1, are the primary means for meeting this requirement in the DRTSJ.

Finally, real-time — especially non-trivial distributed real-time — systems have a unique need for flexible, application-defined resource management. The RTSJ is extended in the DRTSJ by providing a scheduling framework. Software designers may provide user-level application-specific policies to govern the local and distributed scheduling of activities in the system. These policies may range from the RTSJ default of priorities, to deadline-based policies (not only the familiar "earliest deadline first," but also ones widely used outside the classical hard real-time community, such as "minimize the number of missed deadlines," "minimize mean tardiness," etc.), to time/utility function utility accrual based policies [25].

The EG expects to release a series of Early Draft Review (EDR) packages to share our emerging vision of the DRTSJ with the JCP members and the wider community. We seek feedback from researchers and practitioners to leverage their perspectives on distributed real-time programming, and to ensure that emphasis is placed on problems for which there is a need for solutions.

3. DISTRIBUTABLE THREADS

Many distributed systems have a natural expression as a collection of concurrent sequential flows of execution within and among objects. The *distributable thread* programming

model supported in OMG's recent Real-Time CORBA 1.2 standard (abbreviated here as RTC2) [23] provides such threads as first-class abstractions. Distributable threads first appeared as "distributed threads" in the Alpha OS kernel [19, 12]. Subsequently the Alpha and Mach 3 microkernels were merged by the Open Software Foundation as the basis for its MK7.3 OS [24], which (together with some other research OSs) substituted the (less accurate) term "migrating threads."

A distributable thread is a single thread of execution with a globally unique identifier that extends and retracts through local and remote objects. Thus, a distributable thread is an end-to-end control flow abstraction, with a logically distinct locus of control flow movement within and among objects and nodes, that directly manifests the distributed behavior of many systems. We will refer to distributable threads as *threads* except as necessary fo clarity.

Threads extend and retract across object boundaries by performing remote procedure calls (RPCs). Therefore, the distributable thread may be seen as a chain of local threads (or *segments*) connected by intervening RPCs. A thread's stack is thus distributed across the set of nodes hosting segments at any given time. While the synchrony of a conventional method invocation is often cited as a concurrency limitation [27], a distributable thread is sequential rather than synchronous (send/wait). It is always executing somewhere (unless it is "in-flight" during an RPC communication step), while it is the most eligible there – it is not doing send/waits as with RPC. Each nodes processor is always executing the most eligible distributable thread present; the other distributable threads wait as they should. Remote invocations and returns constitute scheduling events at both source and destination nodes, and may be dealt with accordingly by the active scheduling policy.

Because Java is an aggressively multi-threaded language, it is intuitive for programmers to express applications in terms of this type of concurrency.

A distributable thread may have end-to-end time constraints. Typically, a time constraint is declared as a lexically scoped attribute of an action performed by a schedulable entity (e.g., thread). While executing within the time constraint scope, the thread might be said to be a "real-time" one, and otherwise a "non-real-time" one. When a remote invocation occurs, the platform causes a time constraint to be propagated to, and enforced on, any nodes downstream of the declaration point.

3.1 Partial Failures

Non-trivial dynamic distributed systems must be presumed always to be partially failed. At any given time, transmissions are being lost for a variety of reasons, nodes are overloaded, failing, rebooting, joining, or departing the system. Because distributable threads execute in an environment subject to partial failures typically not experienced by nodal (e.g., operating system) threads, provisions must be made to ensure the end-to-end integrity of distributable threads in a manner that assigns responsibilities appropriately between the particular system and its applications, and can be reasoned about. (The Real-Time CORBA specification leaves this issue to be addressed as added value by the ORB vendors.)

Several approaches to distributable thread integrity have been demonstrated in prior work. The DRTSJ seeks to build on this prior work 1) by providing example implementations of those existing schemes; 2) by providing APIs allowing applications to use their own integrity policies; and 3) by providing a set of example implementations from the cutting edge of research in this area, focused on thread integrity in the increasingly important field of mobile, ad-hoc networks.

3.2 Implementation Implications

Implementers are able to implement distributable threads with any transport infrastructures (RMI, SOAP, Real-Time CORBA, etc.) that provides an interface comparable to that of Java RMI's programming model. The DRTSJ distributable threads abstraction is consciously very similar to that of Real-Time CORBA 2, to facilitate application programmers' ability to write distributed programs that use both infrastructures, as has been requested by numerous prospective DRTSJ users.

4. EARLY DRAFT REVIEW #1 SPECIFICATION

As mentioned above, the JSR-50 EG will release one or more JCP 2.6 Early Draft Reviews for community feedback. The initial EDR is focused on distributable threads. Some details about EDR1 are below. The RI differs from (is more speculative than) the specification document for the reasons described earlier. Note that, per JCP 2.6 [33, Section 3.3], an EDR does not necessarily constitute a commitment about the eventual final specification.

4.1 Modifications to Java and RTSJ Classes

It is an explicit goal of the DRTSJ project to minimize modifications to the syntax or semantics of Java or RTSJ facilities. In some cases, modifications have been unavoidable. Furthermore, extensions which may not be entirely natural have been made in order to minimize the effects on the underlying platform. The DRTSJ is considered a superset of the RTSJ specification, and therefore includes by reference all modifications specified in the RTSJ Specification [3, "Standard Java Classes"]. We summarize those modifications here:

Marking Classes Serializable: The following classes have been marked `Serializable` in order to facilitate distributed behaviors: `HighResolutionTime` and `PriorityParameters`. In addition, a subset of the exceptions introduced by the RTSJ will be marked Serializable.

Behavior of Java Monitors: Java monitors must respect distributable thread identity. It is left to the JVM implementer to determine the best mechanism to enforce this; the DRTSJ Reference Implementation relies on a 1:1 mapping of distributable threads to local surrogate threads in order to

The consequence of this is that existing RMI style applications may exhibit different behavior. Deadlock situations such as those described in [36, Section 5.3], [34], and [10] are mitigated. Others have considered alternative approaches such as bytecode manipulation [37].

Distributable Threads are not java.lang.Threads: `Thread.currentThread()` and `RealtimeThread`'s equivalent do not return the current `DistributableThread`. Rather, they return the surrogate thread which represents the distributable thread's current segment. The behavior of meth-

ods provided by the surrogate threads is currently undefined in the context of a distributable thread; proxy methods are provided by the DistributableThread class.

4.2 Scheduling

No additional specification is made regarding the implementation of particular schedulers in the DRTSJ. Like the JSR-1 EG, we expect that distributed real-time systems designers will require a variety of advanced scheduling algorithms, and that vendors will provide them. The RTSJ EG expressed its intent for `Schedulable`s in [7]; we support this requirement and provide a candidate scheduling framework in the DRTSJ RI. A major difference is that the DRTSJ RI scheduling framework allows user level scheduling algorithms, not just algorithms supplied by a JVM vendor.

We extend RTSJ's requirement for a `PriorityScheduler` to accommodate RT CORBA's *Case 2* distributed scheduling model. In the case of a thread whose scheduler is set to the priority scheduler, a DRTSJ implementation must propagate the currently-active `PriorityParameters` as the thread extends and retracts through distributed objects. This is identical to RT CORBA's *client propagated Priority Model*. In the event that a distributable thread attempts to make a remote invocation with non-remote, non-serializable scheduling parameters, an appropriate `RemoteException` is raised.

No interfaces for conducting distributed scheduling are defined, apart from the remote methods exposed by the `DistributableThread` class.

4.3 Threads

The DRTSJ specifies a new class called `DistributableThread`, which represents a thread (potentially) spanning multiple nodes. This object is not descended from `java.lang.Thread` because it requires extended semantics which would undermine the class definition. A `DistributableThread` is a remote object, in the RMI sense. The implementation object must be created and exported on the distributable thread's root node. The means by which mutations to the object on the root node are propagated to other segments of the thread is a quality of implementation issue unless explicitly described in the specification.

As discussed above, distributable threads are subject to partial failures as a consequence of their execution environment. Implementers must provide a thread integrity mechanism which seeks to ensure that they maintain consistent state: that threads must have only one active head (execution point) at any given time; that portions of threads which are deemed to have failed receive exceptions or other notification which will allow the application to gracefully respond and clean up from these conditions. The precise number and nature of integrity mechanisms is left unspecified, however the topic has been well studied by us and others [9, 6, 5] and implementers will be able to draw on prior research and development to make integrity mechanisms a key differentiator for their products.

4.4 Memory Management

The DRTSJ Expert Group has investigated the utility of remote memory areas and is of the opinion that such a facility is unlikely to be used. As such, EDR#1 includes no facilities for providing control over remote memory – e.g., a distributed way to name a scoped area from off the node. In any case, it is difficult to make sense of this semantically.

EDR#1 is currently silent about how distributable threads interact with memory management. The RTSJ community seems not yet to have fully converged on solutions to issues with RTSJ scoped memory, even in the latest versions of the RTSJ. When consensus emerges there, we anticipate addressing that viewpoint from a distributed system standpoint.

4.5 Asynchrony

Semantically sensible approaches to distributed events are still under investigation. The EG has made no commitment yet about their appearance in the DRTSJ specification document or RI. As such, no distributed extensions of `AsyncEvent`s will appear in EDR#1. However, asynchronous events are heavily used in the local implementations of scheduling framework primitives. Implementation experience so far leads us to believe that distributed events will not prove to be a useful complement to the threads model. Because RTSJ's `AsyncEventHandlers` are intended to be lightweight and failure-free, we propose that programmers who require distributed behaviors inside event handlers spawn or signal a full thread to create the effect. A low-level distributed events facility is expected to appear in the DRTSJ RI for thread control, failure notification, time constraint changes, and other state changes of interest to thread heads (execution points). No decision has yet been made regarding the visibility of this facility to application code.

A thread's "interruptible" status propagates along with the thread's head across node boundaries in a manner consistent with that prescribed in the RTSJ specification, with the caveat that primitives for distribution including all methods on instances of `RemoteObject` are not interruptible. In EDR#1, no guarantee is specified about the timely delivery of an interruption to the head of the target thread.

5. EARLY DRAFT REVIEW #1 SOFTWARD SUITE

The EDR#1 software suite consists of a modified RTSJ-compliant J2ME virtual machine and a class library consisting of modified RTSJ classes as well as new classes in the `javax.drealtime` package. In addition, a tested real-time Linux configuration and demonstration application are provided.

5.1 Real-Time RMI

The DRTSJ RI provides a full Java Remote Method Invocation (RMI) stack, called RT-RMI, intended for use in RTSJ virtual machines. RT-RMI is wire-protocol compatible with Sun's JDK1.4 RMI implementation, while providing extended wire protocols for invocations between DRTSJ-compliant JVMs. A datagram-based RMI wire protocol with application-level reliability mechanisms has been provided in order to demonstrate and test DRTSJ applications in dynamic and mobile, ad-hoc networks where TCP is a poor engineering solution.

RT-RMI has organic support for carrying arbitrary invocation contexts across nodes, facilitating the construction of distributable threads and potentially other end-to-end programming abstractions. By default, invocations between RT-RMI-capable nodes carry their execution context with

them. Behaviors analogous to Level 2 Integration as discussed in [36] are provided transparently across the system.

The Sun Microsystems RMI wire protocol specification [31] relies heavily on Java Object Serialization [30]. However, other implementations such as RMI-IIOP [32] provide their own object marshalling facilities. The EDR#1 specification requires the Java RMI programming model, but has been decoupled from particular Java RMI implementations. Real-time object serialization and the accompanying memory allocation behaviors are left as a quality of implementation issue.

RT-RMI as provided in EDR#1 defines several new exceptions subclassed from `RemoteException` and `RuntimeException` to indicate failure conditions resulting from violations of end-to-end time constraints or thread integrity events. These exceptions should be caught and dealt with by application code; however the safety and consistency of the distributed system is preserved even if the application fails to deal with these events.

5.2 Distributable Threads

The DRTSJ implementation team has implemented distributable threads capable of interoperating with various JVMs and transport infrastructures. These threads are capable of traversing nodes with standard, RTSJ, and DRTSJ virtual machines, yielding the best available timeliness behavior feasible on each participant. Invocations and returns are presented to the programmer in a manner congruent with the RMI programming model, but have been decoupled from particular RMI implementations to the extent possible.

5.3 Thread Integrity

The DRTSJ RI provides: example implementations of prior art integrity (e.g., orphan detection and elimination) policies; APIs allowing applications to provide their own integrity policies; example implementations of new research focused on thread integrity in mobile, ad-hoc networks. We refer to the class of thread integrity protocols implemented to date as thread maintenance and repair (TMAR) protocols.

The following protocols have been implemented and will appear in the preliminary DRTSJ RI:

- *Thread Polling*, a protocol originally implemented in the Alpha research OS kernel
- A *fast failure detector* (FFD) driven TMAR, which detects link failures immediately and triggers orphan cleanup in the event of down/upstream failures. This policy provides best-effort ordered orphan cleanup[1] if requested
- *TPR*, an approach which provides deterministic detection and cleanup times for failed distributable threads with failure handlers [6]
- *D-TPR*, an evolving algorithm and protocol for predictable detection and cleanup times in wireless and dynamic networks [5]
- *W-TPR*, an evolving algorithm and protocol for predictable detection and cleanup times in wireless and dynamic networks [5]

In addition, we expect to implement *Node-Alive* [9], a more conservative approach targeted for local area networks and very high reliability.

5.4 Pluggable Scheduling

Implementations of example distributed real-time applications and high-quality thread integrity mechanisms require support from scheduling policies. To facilitate experimentation and the construction of an acceptable RI, the implementation team has included an optional *Metascheduler* component, allowing arbitrary user-defined scheduling disciplines to be defined. While the RTSJ does specify interfaces which schedulers and schedulable objects must meet, it does not provide the primitives necessary to implement scheduling policies without the cooperation of the RTJVM vendor [7, 38].

The *Metascheduler* implements an abstract scheduling framework intended to support pluggable schedulers consistent with the RTSJ vision. While EDR#1 does not yet offer a proposed API, the framework and *Metascheduler* are included in the RI.

A variety of scheduling disciplines have been implemented, ranging from simple, traditional (e.g., fixed priority, EDF), to *Time-Utility Function/Utility-Accrual* (TUF/UA) policies. In particular, we demonstrate a combined TUF/UA scheduling and thread integrity mechanism for providing bounded-time, end-to-end thread failure detection and recovery. [6]

The scheduling framework and *Metascheduler* is inspired primarily by prior work in scheduling frameworks in the Alpha research OS kernel [4], the Open Group Research Institute Mk7.3a OS integrated Alpha/Mach kernel [24], and in particular the local [18] and distributed threads [17] Metascheduler work at Virginia Tech.

5.5 The DRTSJ RI Distribution

The DRTSJ RI and TCK will be delivered in two forms: First, traditional tarball and JAR files appropriate for cross-platform evaluation and use by JCP members; second, because of the complexity and inherent dependencies, a set of Debian packages is being maintained to streamline the "getting started" process. A package repository containing the DRTSJ core libraries, with references to all required dependencies will be provided, and constructing a test system will simplified to a single Debian "`apt-get`" command.

It is possible to compose and interact with DRTSJ applications without a DRTSJ-compliant JVM. The distributable threads abstraction will run atop a vanilla JVM using `java.lang.Thread` sections, and the programmer may specify if transitions should be made to full, real-time `DistributableThread` sections upon arrival in DRTSJ-compliant JVMs.

The DRTSJ EDR#1 RI consists of

- a set of class libraries implementing the DRTSJ APIs
- a set of external dependencies, including the Apache build environment and a Debian Linux system with Linux Kernel 2.6, patched with the most recent real-time extensions

[1] There is some confusion regarding the definition of *best-effort*. Here we use it in the conventional sense: The TMAR protocol attempts to provide ordered cleanup within a reasonable time constraint, but no guarantees are made to the application.

The JSR-50 proposal reserved the right to require modest changes to underlying RTJVMs. While the implementation team is not certain what changes will be required by the final DRTSJ, the current EDR#1 RI does require specific changes. These changes are included in an off-the-shelf binary DRTSJ-compatible JVM product from Apogee Software, Inc., included as part of the EDR#1 suite.

5.6 Demonstration Application

Virginia Tech has written a demonstration application [1] to help prospective users better understand the DRTSJ. The demonstration application is also providing essential feedback to the team designing and implementing the DRTSJ and RI. That work was performed in an ONR-funded Advanced Wireless Integrated Navy Network (AWINN) project at Virginia Tech. [2] Both the AWINN project and MITRE's DRTSJ focuses are on mobile, ad hoc wireless networks (MANETs) with end-to-end time constraints.

The demonstration consists of a coastal air defense simulation, a non-trivial application written on the EDR#1 Reference Implementation, using distributable threads as the end-to-end programming and scheduling abstraction. The application consists of a collection of distributed components for managing on-board sensors, fighter/interceptors, tracking systems, and command and control C2 operations in a multi-ship naval warfare simulation. The simulation testbed includes thirteen nodes comprising a scenario generator, a MANET/dynamic network simulator, and four communications/routing nodes, and seven application nodes running DRTSJ application code atop Linux 2.6 with real-time extensions. Novel approaches to enforcing distributable thread integrity are demonstrated and evaluated against mission metrics.

The demonstration currently exercises TMAR protocols and accompanying scheduling algorithms which provide probabilistic timing assurances for end-to-end thread behavior in the presence of application- and MANET-induced run-time uncertainties. These uncertainties include those induced by workloads, node/link failures, message losses, and node membership changes (previously open problems).

This demonstration application will be provided with the EDR#1 RI in order to aid first-time users and illustrate how a non-trivial system may be constructed using the DRTSJ.

5.7 Virtual Machine Support

Apogee Software, Inc. has agreed to incorporate DRTSJ EDR#1-specific changes into their RTSJ-compliant Aphelion Java runtime environment product. These changes include modifications to the class library in support of the DRTSJ specification as well as more aggressive changes to support experimental work on advanced pluggable and distributed scheduling policies in the RI.

For example, the modified version of Apogee's Aphelion RTJVM has hooks for notifying user-space schedulers of state changes in Java object monitors. This is a key enabler for implementations of RTSJ `Scheduler` implementations in pure Java. While the trial implementation is not yet performant, it is sufficient for demonstrating the behaviors required for accurate and safe scheduling of distributable real-time threads.

6. THE WAY AHEAD

Work on JSR-50 is focused on delivering Early Draft Release #1 by late Fall of CY2006. The current implementation schedule has two primary goals: first, the delivery of a JSR-50 submission; second, the research and development needs of the team members at MITRE and Virginia Tech. The development schedule is always subject to change due to staffing conflicts (students academic responsibilities always have highest priority, and MITREs sponsor needs always have highest priority) and other unforeseen circumstances.

The DRTSJ project is currently understaffed – our immediate objective is to reach a first Early Draft Release for a document and RI (plus a compatible JVM) that meets our own needs, and – very importantly – is sufficient to engage additional contributors to further the concepts and implementation. The schedule and content of future draft releases and the final submission depend on the quality and quantity of design and especially implementation participation from the broader community.

7. ACKNOWLEDGEMENTS

The work described herein has been sponsored by the MITRE Corporation, the US Air Force Electronic Systems Center, the US Navy Office of Naval Research through the AWINN grant program, and Virginia Polytechnic and State University. Particular thanks are due Apogee Software for their generous donation of engineering support for their DRTSJ-compliant Aphelion Real-Time JVM.

8. REFERENCES

[1] J. Anderson and B. Ravindran. AWINN task 2.2 final demonstration: A coastal air defense scenario. [Presentation to USN Office of Naval Research, August 2006], August 2006.

[2] Advanced Wireless Integrated Navy Network (AWINN) homepage. http://awinn.ece.vt.edu.

[3] G. Bollella, B. Brosgol, P. Dibble, S. Furr, J. Gosling, D. Hardin, M. Turnbull, R. Belliardi, D. Holmes, and A. Wellings. The real-time specification for Java (version 1.0.2). Specification JSR-1, Java Community Process, 2006. Available: http://www.rtsj.org/specjavadoc/book_index.html.

[4] R. K. Clark, E. D. Jensen, and F. D. Reynolds. An architectural overview of the Alpha real-time distributed kernel. In *Proceedings of the USENIX Workshop on Microkernels and Other Kernel Architectures*, April 1992.

[5] E. Curley. Integrity assurances for distributable real-time threads in dynamic networks. Master's thesis, Virginia Polytechnic and State University, September 2006. [Anticipated].

[6] E. Curley, J. Anderson, B. Ravindran, and E. D. Jensen. Recovering from distributable thread failures with assured timeliness in real-time distributed systems. In *Proceedings of the 2006 SRDS*, October 2006. [To Appear] Available: http://www.real-time.ece.vt.edu/srds06.pdf.

[7] P. Dibble and A. Wellings. The real-time specification for Java: Current status and future work. In *Proceedings of the Seventh IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 71–77, May 2004.

[8] DRTSJ public web site. http://drtsj.org.

[9] J. Goldberg, I. Greenberg, R. K. Clark, E. D. Jensen, K. Kim, and D. M. Wells. Adaptive fault-resistant systems (chapter 5: Adaptive distributed thread integrity). Technical Report csl-95-02, Computer Science Laboratory, SRI International, Menlo Park, CA., January 1995. http://www.csl.sri.com/papers/sri-csl-95-02/.

[10] B. Haumacher, T. Moschny, J. Reuter, and W. F. Tichy. Transparent distributed threads for Java. In *Proc. 5th International Workshop on Java for Parallel and Distributed Computing in conjunction with the International Parallel and Distributed Processing Symposium (IPDPS 2003)*, 2003.

[11] E. D. Jensen. Rationale for the direction of the distributed real-time specification for Java panel position paper. In *Proceedings of the Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, 2002*, pages 258–259, 2002.

[12] E. D. Jensen and J. D. Northcutt. Alpha: A non-proprietary operating system for large, complex, distributed real-time systems. In *IEEE Workshop on Experimental Distributed Systems*, pages 35–41, 1990.

[13] JSR-1 Expert Group. JSR-1 proposal: Real-time specification for Java. http://jcp.org/en/jsr/detail?id=1.

[14] JSR-282 Expert Group. JSR-282 proposal: Real-time specification for Java version 1.1. http://jcp.org/en/jsr/detail?id=1.

[15] JSR-50 Expert Group and E. D. Jensen. JSR-50 proposal. http://jcp.org/en/jsr/detail?id=1.

[16] B. W. Lampson. Hints for computer system design. In *SOSP '83: Proceedings of the ninth ACM symposium on Operating systems principles*, pages 33–48, New York, NY, USA, 1983. ACM Press. Available: http://research.microsoft.com/~lampson/33-Hints/WebPage.html.

[17] P. Li, B. Ravindran, H. Cho, and E. D. Jensen. Scheduling distributable real-time threads in Tempus middleware. In *IEEE Conference on Parallel and Distributed Systems*, pages 187 – 194, July 2004.

[18] P. Li, B. Ravindran, et al. A formally verified application-level framework for real-time scheduling on POSIX real-time operating systems. *IEEE Trans. Software Engineering*, 30(9):613 – 629, Sept. 2004.

[19] J. D. Northcutt. *Mechanisms for Reliable Distributed Real-Time Operating Systems — The Alpha Kernel*. Academic Press, 1987.

[20] Object Management Group. Dynamic scheduling real-time CORBA 2.0 (joint revised submission), 2001. orbos/2001-04-01 ed.

[21] Object Management Group. Extensible transport framework specification – final adopted specification, 2004. ptc/04-03-03 ed.

[22] Object Management Group. Data distribution service for real-time systems, v1.1, 2005. formal/2005-12-04.

[23] OMG. Real-time CORBA 2.0: Dynamic scheduling specification. Technical report, Object Management Group, September 2001. OMG Final Adopted Specification, http://www.omg.org/docs/ptc/01-08-34.pdf.

[24] Open Group Research Institute's Real-Time Group. *MK7.3a Release Notes*. The Open Group Research Institute, Cambridge, Massachusetts, October 1998. Available: http://www.real-time.org/docs/RelNotes7.Book.pdf.

[25] Virginia Tech real-time laboratory publications site. http://www.real-time.ece.vt.edu/papers.html.

[26] RTSJ public web site. http://rtsj.org.

[27] U. Saif and D. J. Greaves. Communication primitives for ubiquitous systems or RPC considered harmful. In *21st International Conference on Distributed Computing Systems Workshops (ICDCSW '01)*, 2001.

[28] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Trans. Comput. Syst.*, 2(4):277–288, 1984.

[29] Sun Microsystems. Java™ message service specification final release 1.1, April 2002. Available: http://java.sun.com/products/jms/.

[30] Sun Microsystems. Java™ object serialization specification. Technical report, Sun Microsystems, 4150 Network Circle, Santa Clara, CA, November 2002. [Revision 1.4.4] Available: http://java.sun.com/j2se/1.4/pdf/serial-spec.pdf.

[31] Sun Microsystems. Java™ remote method invocation specification. Technical report, Sun Microsystems, 4150 Network Circle, Santa Clara, CA, November 2002. [Revision 1.9, Java™ 2 SDK SE, v.1.4.2] Available: http://java.sun.com/j2se/1.4/pdf/rmi-spec-1.4.2.pdf.

[32] Sun Microsystems. Java™ RMI over IIOP technology documentation home page. Technical report, Sun Microsystems, 4150 Network Circle, Santa Clara, CA, November 2002. [From J2SDK 1.4.2 Release Notes] Available: http://java.sun.com/j2se/1.4.2/docs/guide/rmi-iiop/index.html.

[33] Sun Microsystems. Jcp2: Process document, v2.6, March 2004. Available: http://jcp.org/en/procedures/jcp2.

[34] E. Tilevich and Y. Smaragdakis. Portable and efficient distributed threads for Java. In *Conf. Proc. Middleware'04 conference*, October 2004.

[35] J. Waldo, G. Wyant, A. Wollrath, and S. Kendall. A note on distributed computing. Note SMLI TR-94-29, Sun Microsystems Laboratories, Inc., 2550 Garcia Avenue, Mountain View, VA 94043, November 1994.

[36] A. Wellings, R. K. Clark, E. D. Jensen, and D. Wells. A framework for integrating the Real-Time Specification for Java and Java's remote method invocation. In *Proc. of the 5th IEEE International Symposium on Object Oriented Real-Time Distributed Computing*, April 2002. Available: http://www.real-time.org/docs/isorc02_v41.pdf.

[37] D. Weyns, E. Truyen, and P. Verbaeten. Distributed threads in Java. In *Proceedings of the International Symposium on Distributed and Parallel Computing, ISDPC 2002*, 2002.

[38] A. Zerzelidis and A. J. Wellings. Getting more flexible scheduling in the rtsj. In *Proceedings 9th IEEE ISORC*, pages 3–10. IEEE Computer Society TC on Distributed Processing, IEEE Computer Society, April 2006.