

The Distributed Real-Time Specification for Java

An Initial Proposal

E. Douglas Jensen
The MITRE Corporation
jensen@real-time.org

Abstract

Work is beginning on the Distributed Real-Time Specification for Java, as part of Sun's Java Community Process. This paper summarizes some ideas about an initial approach to the specification. The approach is based on providing a natural and minimal mechanistic extension to Remote Method Invocation (RMI) to support the end-to-end timeliness (and other) properties of distributed – in the sense of trans-node – behaviors. These timeliness properties must be preserved for any distributed real-time computing system, regardless of its application programming model – whether RPC, mobile objects, or whatever. The proposed extension also facilitates real-time distributed programming with control flow programming models in particular. A similar facility has proven effective in several other distributed real-time operating system and middleware contexts, and is a primary feature of the unified proposal to OMG for Dynamic Real-Time CORBA.

1. Introduction

The Java platform has significant potential value in the real-time space – both as a productive programming language, and as a portable applications platform on the plethora of real-time operating systems and processors.

Recognizing this, a group of real-time and Java experts convened a series of NIST-sponsored workshops beginning in June 1998 to develop requirements for “real-time Java” [Carnahan and Ruark 99]. A major problem the group encountered when defining the phrase “real-time Java” was that the term “real-time” itself is used in both the practitioner and research communities with widely varying, usually ineffective, and even contradictory, definitions – likewise for concomitant fundamental terms such as “hard,” “soft,” “predictable,” and “deterministic” (see, for example, the archives of the news-

group comp.realtime). The NIST group extensively discussed these and other terms needed for crisply specifying requirements for real-time Java, and adopted a relatively rigorous lexicon in “Section 1 Concepts” of [Carnahan and Ruark 99].

Now work has begun on the Distributed Real-Time Specification for Java, for which that lexicon is even more important, and so a brief synopsis of it is provided in Section 2 of this paper.

The term “distributed” also lacks a widely accepted and precise definition. This paper neither needs nor attempts to remedy that, but instead employs a very simple classification of “distributed” systems in the Section 3 that seems sufficient to at least begin considering meaningful interpretations of the phrase “distributed real-time specification for Java.”

A specific proposal for an initial approach to the Distributed Real-Time Specification for Java (under Sun's Java Community Process [Sun 98]) is outlined in Section 4. Concomitant facilities implied by the approach proposed here (at least for certain control flow programming models), but not necessarily recommended to be part of the initial specification, are summarized in Section 5.

Some previous work related to this proposal is identified in Section 6.

2. Real-Time Concepts and Terms

Popular use of most fundamental real-time terms, and thus the concepts underlying them, is often confused and even erroneous. To avoid the negative consequences that problem has historically imposed, a coherent real-time lexicon was adopted by the NIST-sponsored real-time Java requirements workshop series (see “Section 1 Concepts” of [Carnahan and Ruark 99]). That lexicon is even more important to the Distributed Real-Time Specification for Java, because most traditional real-time concepts and terms do not scale up to distributed

systems. Thus, the lexicon is summarized in this section.

2.1 Time Constraints

A real-time application includes actions whose completions are time-constrained. These time constraints are inherent in the behavior of the application and its execution environment. The most common action completion time constraint is a deadline, but there are many other kinds.

A *time constraint* is a lexically scoped attribute of a computational entity (say, thread). Each time-constrained action is performed by a thread executing code that includes a time-constrained block of instructions called the time constraint's *scope*.

The timeliness of a thread's transit through that scope depends on the time at which the thread's execution point reaches the end of the scope. While a thread is executing within a time constraint scope, it is a "real-time thread," otherwise, it is a "non-real-time thread." The passage of a thread's execution point into or out of a time constraint scope is a *scheduling event* – a decision must be made about which thread should be executing.

A *deadline* time constraint specifies that the timeliness of a thread's transit through that scope depends on whether the thread's execution point reaches the end of the scope before the deadline time has occurred, in which case the deadline is satisfied.

A *hard deadline* is a time constraint, such that if the deadline is satisfied, then the deadline-constrained portion of the thread's execution is timely; otherwise, that portion is not timely.

A *soft deadline* is a time constraint, such that if the deadline is satisfied, then the deadline-constrained portion of the thread's execution is more timely; otherwise, that portion is less timely.

A *general time constraint* is any relationship between the time when the thread's execution point reaches the end of that time constraint's scope, and the utility to the system of when it does so. All deadline time constraints are special cases of a general time constraint.

A thread for which there is no relationship between when it completes execution and its utility to the system is a *non-real-time* one.

2.2 Scheduling

Multiple time-constrained threads generally con-

tend for resources – notably processor cycles, but also other exclusively shared resources, both physical (e.g., communication paths) and logical (e.g., synchronizers). This contention must be resolved into a sequence of resource accesses – e.g., thread executions. In general, contention for all shared resources should be resolved in a consistent manner.

Thread *scheduling* is deciding in what order they all will execute. Each time thread scheduling is performed, it establishes a sequence – a *schedule* – for all threads ready at that time. Scheduling may be performed statically (prior to execution time) by a person or a program, or dynamically (at execution time) by a user or the system software.

Thread *dispatching* is granting resource access – e.g., launching – the currently most eligible thread. When scheduling is employed, dispatching occurs in schedule order.

Thread scheduling is not always necessary or computationally feasible – dispatching alone may be sufficient.

Moreover, some actions are never threads and thus not schedulable – most commonly, interrupt service routines and certain OS services, which execute either when invoked or automatically as needed (other OS services are scheduled in concert with application entities). These actions do not have time constraints; instead, their timeliness may be characterized in terms of tight upper bounds on completion time.

Dispatching when scheduling is not employed establishes a thread resource access – e.g., execution – sequence one thread or non-schedulable action at a time.

A *scheduling event* or *dispatching event* (in general, a *sequencing event*) is an event – e.g., a thread becoming ready or blocked, contending for a resource, etc. – that may cause the execution sequence to change.

2.3 Sequencing Optimization Criteria

Contention for execution (and all other sequentially shared physical and logical resources) usually should be resolved according to an application-specific policy that seeks maximal usefulness as measured by a *sequencing optimality criterion*.

In real-time computer systems, this criterion has two dimensions.

The first dimension is optimality or satisficing of *collective timeliness* – i.e., the resulting timeliness of all those threads, with respect to an application-specific criterion. Every possible sequence of thread executions has an optimality value according to this metric. Optimum collective timeliness is a rare special case.

Hard and soft real-time are distinguished by their sequencing optimality criteria. Hard real-time has only one timeliness component in its collective timeliness optimality criterion: always meet all hard deadlines. Soft real-time sequencing includes all other possible timeliness components in sequencing (usually scheduling) optimality criteria – very common examples are “minimize mean weighted tardiness,” “minimize the number of missed deadlines according to importance,” and “minimize maximum tardiness” [Pinedo 95]. Hard and soft real-time are not distinguished by optimality per se: hard real-time sequences do have maximum optimality in this respect; soft real-time sequences may also, but generally do not.

The second dimension of sequencing optimality criteria is *predictability of collective timeliness* – i.e., predictability of the first dimension.

Informally, a property is *predictable* to the degree that it is known in advance. One end point of the predictability scale is *determinism*, in the sense that the property is known exactly in advance. The other end point can be characterized as *maximum entropy*, in the sense that nothing at all is known in advance about the property. In stochastic real-time systems (which include hard real-time systems as a special case), one well-defined way to measure predictability is coefficient of variation C_v , which is defined as $\text{variance}/\text{mean}^2$. For the deterministic distribution, $C_v = 0$, and the extreme mixture of exponentials distribution is an example of a maximally non-deterministic one whose $C_v = \infty$.

The two dimensions of sequencing optimality criteria are orthogonal. The collective thread sequence optimality may be deterministically optimal (the hard real-time case), deterministically sub-optimal, non-deterministically optimal, and non-deterministically sub-optimal (all soft real-time cases).

The two dimensions generally have to be traded off against one another on an application-specific basis. For example, it may be possible or necessary to choose between two sequences – one of which provides better optimality (e.g., lower maximum num-

ber of missed deadlines) but worse predictability (higher variance of missed deadlines), and the other of which provides worse optimality but greater predictability. Hard real-time is a special case where the two dimensions do not have to be traded off (the collective timeliness criterion of meeting all hard deadlines is deterministically optimum).

Given a collective timeliness sequencing (e.g., a thread scheduling) criterion, a suitable sequencing algorithm is chosen or devised. When scheduling is employed, the algorithm is usually called a *scheduling algorithm*, and when only dispatching is employed, the algorithm is usually called a *dispatching rule*. There may be more than one algorithm for a given criterion – e.g., the hard real-time criterion is met by the earliest deadline first (EDF) algorithm, and by least laxity first algorithm, among others. Conversely, a specific algorithm may be suitable for different criteria – EDF meets the hard real-time criterion, and also satisfies the soft real-time criterion “minimize maximum lateness.”

In principle, the timeliness specifications for non-scheduled entities, especially interrupt service routines, correspond exactly to those for scheduled entities. Hard and soft upper bounds replace hard and soft deadlines, and optimality of collective timeliness and predictability of collective timeliness are unchanged. The difference is that timeliness and predictability are the responsibility of designers and implementers instead of schedulers.

3. Categories of Distributed System Programming Models

For the purpose of this paper, the term *distributed system* informally refers to a computing system whose programming model is based on there being application entities that exhibit trans-node behaviors. Each of these trans-node behaviors has end-to-end properties – these properties may include (but are not limited to) a unique identifier, timeliness, security, resource ownership, etc.

Distributed systems can be categorized in various ways for various purposes; here we categorize them in a very simple way according to their programming model for the trans-node interaction aspect of application behaviors:

- *networked* (asynchronous message passing among objects)
- *control flow* (method invocation between objects)

- *data flow* (e.g., publish/subscribe among objects)
- *blackboards/spaces* (e.g., Linda, JavaSpaces)
- *mobile objects, autonomous agents, autonomous decentralized systems*

A distributed system may provide more than one programming model, but usually only one programming model is the first class abstraction, and the others are implemented in terms of it. For example, OMG's CORBA standard specifies a first class control flow programming model, and an optional data flow programming model as a service layered on top of the control flow model. (Of course, a first class distributed system programming model is normally implemented on a communication facility that typically has multiple levels which are not visible to the application – e.g., a blackboard abstraction may be implemented using RPC that is implemented using asynchronous message passing.)

The first three of these categories of distributed systems have long histories of successful use in real-time as well as non-real-time application domains. The last two categories are gaining interest, but currently have little history of use in real-time domains.

4. An Initial Approach to the Distributed Real-Time Specification for Java

As part of Sun's Java Community Process [Sun 98], Java Specification Request 'JSR-000050 The Distributed Real-Time Specification for Java' [Jensen 99] has been submitted and approved. The Distributed Real-Time Specification for Java will extend the JSR-1 Real-Time Specification for Java [RTEG 00]. This paper elaborates on that JSR-000050 proposal.

The defining characteristic of any real-time distributed computing system, whatever its programming model, is that the end-to-end timeliness (optimality and predictability of optimality) of trans-node application behaviors is acceptable.

In most cases, the fundamental requirement for achieving acceptable end-to-end timeliness is that a trans-node application behavior's timeliness properties – time constraints, expected execution time, execution time received thus far, etc. – be explicitly

employed for resource management (scheduling, etc.) consistently on each node involved in that application trans-node behavior.

Thus, in dynamic real-time distributed systems, these properties must be propagated among corresponding computing node resource managers in operating systems, Java virtual machines (JVM's), middleware, etc. In static real-time distributed systems, these properties can be instantiated a priori; but JSR-000050 the Distributed Real-Time Specification for Java is concerned only with dynamic systems.

In real-time distributed Java systems, a trans-node application behavior's end-to-end timeliness (and other) properties must be acquired, propagated, and deposited when RMI's and associated returns occur.

Enhancing RMI to do that is the primary objective of JSR-000050 the Distributed Real-Time Specification for Java. This enhancement is an appropriate mechanism, regardless of what programming model semantics are manifest with RMI's – e.g., whether performing a basic RPC-like method invocation, or passing an object by copy or reference.

A secondary objective of JSR-000050 is apply the RMI enhancement mechanism to support control flow programming models.

More specifically, control flow in this proposal means an individual program can be distributed in the sense that one or more of its computational entities can execute across physical node boundaries, instead of being confined to a single node or a single object instance on a node, as a conventional thread is. In this paper, we use the term *activity* for such a trans-node computational entity in a distributed program. There may be multiple concurrent activities in a distributed program.

An activity can extend and retract its locus of execution point movement among a program's constituent methods in object instances that may be dispersed across a multiplicity of physical computing nodes, by location-independent method invocations and returns. Within an object instance, the flow of control is normal local thread execution – i.e., activity programming models are generally not intended to be completely location-independent, due to complexities of physical dispersal.

In Java, an activity would be implemented by the concatenation of local (per node) threads sequentially performing RMI's when they transit nodes.

Each activity may have one or more execution scheduling attributes – e.g., priority, time constraints such as deadlines or utility functions, importance – that specify the acceptable end-to-end timeliness for it completing the sequential execution of methods in object instances that may reside on multiple physical nodes. The semantics of acceptability with respect to these end-to-end timeliness attributes is defined by the application, in the context of the scheduling policy being used. Execution of the activity is governed by those scheduling parameters, regardless of the activity's execution point transiting nodes.

There are several reasons why activity-based programming models are worthy of explicit support by the Distributed Real-Time Specification for Java.

First, real-time control flow based on RMI is a natural, well-understood, incremental extension to the Real-Time Specification for Java. Familiarity is an especially important factor in attaining adoption in real-time application domains, such as industrial automation, defense, and telecommunications.

Second, activities as described here correspond to the intent of the activities suggested in the Real-Time CORBA 1.0 standard, and are exactly the same as the activities explicitly defined in the Dynamic Real-Time CORBA joint initial proposal [OMG 99b]. That latter correspondence may facilitate building, programming, and using distributed real-time Java/ CORBA systems.

Third, data flow abstractions can be built cost-effectively using control flow abstractions (as is currently done with CORBA and COM), while the converse is semantically difficult, especially in real-time contexts. (Real-time data flow and other programming models could be the subject of subsequent JSR's.)

Activity-based control flow style distributed programming models are applicable to real-time systems which are hard or anywhere else on the predictability continuum (in the technical sense of Section 2). That continuum is orthogonal to application time-frame magnitudes, which range in practice from microseconds to megaseconds. The approach advocated in this paper is intended to address application timeliness requirements everywhere in that two-dimensional predictability/time-frame space of distributed real-time Java systems.

5. Concomitant Facilities

Activity-based programming models imply the need for a number of concomitant facilities; these programming models can be differentiated by which of these facilities they provide, and the approaches employed to provide them. Not all, or any, of these facilities need be included in the initial Distributed Real-Time Specification for Java – they could be the topic of subsequent specifications, or vendor-specific added value class libraries. They include (but are not limited to) the following.

- Some asynchronous events (i.e., changes in system state) of interest to an activity may have to be coordinated with current activity execution. For example, a violated time constraint, or the failure of a node or network path over which an activity is extended, might require notification of the activity's current execution point. Certain other events that occur at the activity's current execution point – e.g., exceptions and other time constraint expirations – may require the activity to return somewhere back up the invocation chain to execute one or more appropriate handlers (from where it may either continue execution where it left off, or terminate).
- Activity control actions – e.g., suspend, resume, abort, time constraint change, etc. – may have to be propagated to, and carried out at, the activity's current execution point.
- Mechanisms may have to be provided to support maintaining correctness of distributed execution, and consistency of distributed data – in both cases, as defined by the application – for concurrent activities of one or more applications.
- Failures locally perceived by an activity may have to be made manifest to failure suspects for the appropriate set of nodes (in the sense of process groups).

All of these facilities generally would be required to be timely – e.g., subject to completion time constraints.

6. Related Work

The activity entity as described here is based on the *distributed thread*, and associated programming model, abstractions in Jensen's Alpha distributed real-time OS kernel at CMU [Northcutt 87], KSR,

and Concurrent Computer Corporation [Clark et al. 92]. Additional insight into that programming model can be gained from an example application created jointly by CMU and General Dynamics [Maynard et al. 88], and from the collaboration of SRI International, CMU, Connection Technologies, and Digital Equipment on a multilevel secure version of Alpha [Greenberg 93]. Recently this programming model was employed in an advanced technology demonstration of a surveillance tracking system [Wheeler et al. 98] [Clark et al. 99]. The Alpha distributed thread technology was adapted by Ford and Lepreau at the University of Utah to create migrating threads for the Mach 3 kernel [Ford and Lepreau 93] [Ford and Lepreau 94]. Microsoft Research also adapted the Alpha distributed thread model for their Rialto real-time OS [Jones et al. 95]. Sun (apparently independently) devised an RPC implementation technique for their Spring research OS similar to that used under Alpha's distributed threads [Hamilton and Kougiouris 93]. The Alpha distributed thread and associated abstractions and facilities were developed further by The Open Group Research Institute in their MK7.3 OS [Goldstein and Wells 93] [Open Group 98], which is the basis for a forthcoming OS from a major computer vendor (although it seems unlikely that these abstractions and facilities will be exposed in that product). These abstractions and facilities in the MK7 OS were utilized effectively in a large scale, multi-year, U.S. Navy/DARPA technology demonstration leading to the next generation Aegis combat system [NSWC 99]. Digital Equipment and IBM jointly performed additional development of these abstractions and facilities for OS/2 for PowerPC (no public references available). Subsequently, Digital Equipment then Compaq continued this work in the context of a real-time DCOM and a Win32-compatible real-time distributed OS (no public references available). DARPA ITO's Quorum program includes as one of its architectural tenets "path based resource allocation" inspired (in part at least) by distributed threads [Koob 98].

The writers of OMG's Real-Time CORBA 1.0 specification [OMG 99a] came to their own recognition of the value of a distributed thread-like abstraction for real-time distributed object systems. To avoid further delaying an already very late specification, they only included mention of "activities" as a design and analysis concept. The activity abstraction as described in this paper was explicitly incorporated into the subsequent initial joint proposal for Dynamic Real-Time CORBA [OMG 99b].

7. Conclusion

The Real-Time for Java specification by the JSR-000001 Expert Group [Bollella 99] [RTEG 00] already includes asynchronous events and asynchronous transfer of control – essential and difficult facilities needed to support the enhancement to RMI and the activity abstraction as described herein.

The proposed initial approach to the Distributed Real-Time Specification for Java described in this paper was submitted to Sun Microsystems as Java Specification Request (JSR) 000050 [Jensen 99], and after a period of public review, Sun approved it in April 2000. The Expert Group has been formed and is currently writing the specification.

Our tentative goal is for the initial version of specification, if not also of the reference implementation and perhaps the conformance test suite, to be publicly available at JavaOne 2001.

References

Bollella 99 *Real-Time Java: Status and Architecture*, June 1999, <http://www.rti.org/rtas99/rtas.htm>.

Carnahan and Ruark 99 Lisa Carnahan and Marcus 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>, September 1999, National Institute of Standards and Technology.

Clark et al. 92 Clark, R.K., E.D. Jensen, and F.D. 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>, April 1992, USENIX.

Clark et al. 93 Clark, R.K., D.M. Wells, I.B. Greenberg, and E.D. Jensen, *Effects of Multilevel Security on Real-Time Applications*, Proc. Computer Security Applications Conf., Baltimore, December 1993, IEEE.

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

time.org/presentations/wpdrts99/atd/screens.htm, 1998, IEEE.

Clark et al. 99 Clark, R.K., T. Wheeler, P. Hurley, E.D. Jensen, T. Lawrence, A. Kanevsky, J. Maurer, P. Wallace, D.M. Wells, and Y. Zhang, *Application of QoS Driven Adaptive Computing*, Real-Time Systems Symposium Work in Progress, <http://www.real-time.org/papers/rtss98/rtss98.pdf>, <http://www.real-time.org/presentations/rtss98/screen.htm>, 1999.

Ford and Lepreau 93 Ford, B. and J. Lepreau, *Microkernels Should Support Passive Objects*, Proc. I-WOOS, <http://www.real-time.org/papers/passive.pdf>, December 1993, IEEE.

Ford and Lepreau 94 Ford, B. and J. Lepreau, *Evolving Mach 3.0 to a Migrating Threads Model*, Proc. Winter USENIX Conf., http://www.real-time.org/papers/thread_migrate.pdf, January 1994, USENIX.

Goldstein and Wells 93 Goldstein, I. and D.M. Wells, *Alpha/Mach Integration Study*, April 1993, The Open Group (née OSF) Research Institute.

Greenberg et al. 93 Greenberg, I.B., P. Boucher, R.K. Clark, E.D. Jensen, T. Lunt, P. Neuman, and D.M. Wells, *The Secure Alpha Study – Final Summary Report*, March 1993, Computer Science Laboratory, SRI International.

Hamilton and Kougiouris 93 Hamilton, G. and P. Kougiouris, *The Spring Nucleus: A Microkernel for Objects*, Proc. 1993 Summer USENIX Conference, June 1993, USENIX.

Jensen 99 Jensen, E. D., Java Specification Request 000050, <http://java.sun.com/aboutJava/communityprocess/jsr.html>, April 1999, Sun Microsystems.

Jones et al. 95 Jones, M.B., P.J. Leach, R.P. Draves, J.S. Barrera, III, *Support for User-Centric Modular Real-Time Resource Management in the Rialto Operating System*, Proc. of the Fifth Workshop on Hot Topics in Operating Systems, <http://www.real-time.org/papers/rialto.pdf>, May 1995, IEEE.

Koob 98 Koob, G., *Quorum*, <http://www.darpa.mil/ito/research/quorum/index.html>, 1998, DARPA.

Maynard et al. 88 Maynard, D.P., S.E. Shipman, R.K. Clark, J.D. Northcutt, R.B. Kegley, B.A. Zimmerman, and P.J. Keleher, *An Example Real-Time Command, Control, and Battle Management Application for Alpha*, <http://www.real-time.org/papers/gd.pdf>, TR-88121, December 1988, Archons Project, CMU Computer Science Dept.

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

NSWC 99 High Performance Distributed Computing (HiPer-D), <http://www.nswc.navy.mil/hiperd/index.shtml>, 1999, NSWC.

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

OMG 99a *Real-Time CORBA 1.0 Specification*, [ORBOS/99-02-12](http://www.omg.org/ORBOS/99-02-12), and *errata*, [ORBOS/99-03-29](http://www.omg.org/ORBOS/99-03-29), 1999, Object Management Group.

OMG 99b *Joint Initial Proposal for Dynamic Real-Time CORBA*, [ORBOS/99-10-06](http://www.omg.org/ORBOS/99-10-06), October 1999, Object Management Group.

Pinedo 95 Pinedo, M., *Scheduling: Theory, Algorithms, and Systems*, ISBN 0-13-706757-7, 1995, Prentice Hall.

RTEG 00 The Real-Time for Java Expert Group, *The Real-Time Specification for Java*, <http://java.sun.com/aboutJava/communityprocess/first/jsr001/index.html>, 2000, Sun Microsystems.

Sun 98 Sun Microsystems, Java Community Process 1.0, http://java.sun.com/aboutJava/communityprocess/java_community_process.html, December 1998, Sun Microsystems.