NIST Special Publication 500-243

U.S. DEPARTMENT OF COMMERCE

Technology Administration

National Institute of Standards and Technology

Requirements For Real-time Extensions

For the Java™ Platform

Report from the Requirements Group for Real-time Extensions
For the Java™ Platform

Lisa Carnahan, NIST

Marcus Ruark, Commotion Technology, Inc.

Editors

NIST Special Publication500-243

Requirements For Real-time Extensions

For the Java™ Platform

Report from the Requirements Group for Real-time Extensions For the Java™ Platform

Lisa Carnahan, NIST

Marcus Ruark, Commotion Technology, Inc.

Editors

Information Technology Laboratory

National Institute of Standards and Technology
Gaithersburg, MD 20899-0001

September 1999

U.S. Department of Commerce

William M. Daley, Secretary

Technology Administration

Gary R. Bachula

Acting Under Secretary for Technology

National Institute of Standards and Technology

Raymond Kammer, Director

The **Requirements Group for Real-time Extensions to the Java™ Platform** includes representatives from companies and organizations whose expertise spans the computing industry and academia. Industry participants include desktop, server, and enterprise systems providers, embedded systems providers, device manufacturers, and real-time operating system vendors. The goal of the Group is to develop cross-disciplinary requirements for real-time functionality that is expected to be needed by real-time applications written in the Java™ programming language and executing on various platforms. See **http://www.nist.gov/rt-java** for more information regarding the Requirements Group and this document.

Members of the Requirements Group for Real-time Extensions to the Java™ Platform include:

Access Co., Ltd.

Ada Core Technologies, Inc.

Advanced Logic Corp.

Aonix

Aplix Corporation

Apogee Software

Bay Networks / Bay Architecture Lab

Commotion Technology, Inc.

Cornfed Systems, Inc.

Cyberonics

Defense Information Systems Agency/Center for Stds./OSJTF

Enea OSE Systems

Florida State University, Department of Computer Science

Hewlett-Packard Company

Honeywell Inc.

IBM

Insignia Solutions, Inc.

Integrated Systems, Inc.

Intermetrics, Inc. An AverStar Company

Lexmark International, Inc.

Mantha Software, Inc.

Microsoft

Microware Systems Corporation

The Mitre Corporation

Mitsubishi Electric Corporation

Motorola, Inc.

MPI Software Technology

NewMonics, Inc.

NIST/Information Technology Laboratory

Nokia Research

Nortel

NSI Com

Oberon Microsystems

OMRON Corporation

The Open Group

Perennial

Plum Hall, Inc.

Plum Hall Europe, Ltd.

QNX Software Systems, Ltd.

Rockwell Collins

Saville Software, Inc.

Schneider Automation

Siemens AG, A&D

Silverhill Systems, Inc.

SoftPLC Corporation

Sony

SRI International

Sun Microsystems, Inc.

TeleMedia Devices, Inc.

The MITRE Corporation

The Open Group

Wind River Systems, Inc.

Xerox Corporation

Table of Contents

Introduction

The Requirements Group for Real-time Extensions for the Java™ Platform

The National Institute of Standards and Technology (NIST) sponsored the Requirements Group for Real-time Extensions to the Java™ Platform. The Requirements Group includes representatives from companies and organizations whose expertise spans the computing industry and academia. Industry participants include desktop, server, and enterprise systems providers, embedded systems providers, device manufacturers, and real-time operating system vendors. The Requirements Group met during a six-month period in a series of open workshops. Additionally, the Requirements Group continued discussions on the rt-java@raleigh.ibm.com mailing list. The goal of the Group was to develop cross-disciplinary requirements for real-time functionality that is expected to be needed by real-time applications written in the Java™ programming language and executing on various platforms. This document is the result of the Requirements Group's efforts.

Document Overview

The requirements contained in this document were developed by group consensus. Therefore the requirements in total do not necessarily represent the opinions of all Requirements Group members or their organizations. In the Requirements Group workshops, consensus was declared only in cases where the agreement was overwhelming. Unless explicitly stated otherwise, consensus was reached on all requirements. In a few instances, a strong consensus could not be reached. Where strong consensus could not be reached, the requirement is marked as *Open — no consensus.*

Content marked as *Rationale* or *Discussion* is not consensus-based content and should not be considered representative of the opinions of the Requirements Group in total. *Rationale* and *Discussion* paragraphs were drafted by members of the Requirements Group and primarily represent the opinions of the respective authors. These paragraphs serve to provide additional information to those applying the requirements to future specifications.

Following this Introductory section is *Section 1 — Terms and Concepts*. The concepts presented in this section provided the basis for real-time theory and application discussions throughout the workshops and on the mailing list. *Section 2 — Java Traits*, highlights the positive aspects of Java that provided the motivation for developing real-time Java requirements. *Section 3 — Guiding Principles*, lists the principles used by the Requirements Group to provide a basis for these requirements. These principles also provide a future vision of real-time Java. *Section 4 — Core Functionality and Profiles*, provides the Requirements Group's insight regarding necessary functionality that should exist in all real-time Java implementations (core functionality) and be defined in a base specification, and how additional functionality could be delineated in additional specifications. *Section 5 — Core Requirements*, defines the set of requirements that the Requirements Group determined must be addressed in a base real-time Java specification. Finally, *Section 6 — Goals and Derived Requirements*, defines an additional set of general Goals and Derived Requirements that should be addressed in either profiles or future generations of the base real-time Java specification.

Section 1 — Concepts and Terms

Real-time Concepts[1]

An initial consensus of the Requirements Group was that the requirements document be capable of specifying Java platform timeliness properties in a manner that is precise and implementation independent—i.e., timeliness (e.g., predictability) properties inherent in the platform specification, rather than the usual measured (or claimed) timing of an implementation. This is an unconventional capability: no commercial real-time OS (and very few custom or even research ones) that we are aware of are based on specifications with such properties. To accomplish this capability, it is necessary to employ well-defined concepts and terminology. This section provides the real-time concepts and terms needed by this document.

The core terms of real-time computing—such as "real-time," "hard real-time," "soft real-time," "determinism," and "predictable"—are popularly used by the real-time practitioner (i.e., vendor and user) community in either undefined or ill-defined and contradictory ways. In that community, there is no consensus on what "hard" real-time means (but people seem to believe they know it when they see it), and "soft" real-time is usually thought to mean "Que sera sera" (a phrase which instead applies to non-real-time systems). This can readily be observed by reading threads related to the topic of what "real-time" means in popular real-time-related newsgroups and the press. The differences in what "real-time" means were readily observed during early Requirements Group meetings, where many members used the core terms of real-time computing (implied and explicit) in various undefined and contradictory ways. This early confusion in the meetings validated the need for these terms and concepts.

In contrast, the real-time computing research community has an exact definition of "hard real-time," which consequently does not intersect with any of the typical practitioner usages. But that research community fails to define "soft real-time" beyond the tautology of "not hard real-time." That community also employs the term "predictability" without precisely defining it, and instead usually erroneously implies that it is a synonym for "deterministic." Also in contrast, the classical scheduling, e.g., job shop scheduling, community has a large body of formal theory for both deterministic and stochastic scheduling. Nearly all of their concepts and terms can be applied directly to real-time computing; and while a few basic ones (such as "deadline") are well known in real-time computer scheduling theory, most are not. Still, real-time computing requires some concepts and terms that are not found in job shop scheduling theory.

1.0 Real-time Computing

Real-time computing involves two aspects related to timeliness: the objective and the means to that objective. The first aspect is the objective: the degree to which an entity (application, system, OS, Java platform, thread, etc.) operates in real-time—i.e., has acceptable timeliness properties according to its timeliness specification.

Achieving an acceptable degree of timeliness may or may not involve the second aspect of real-time computing: real-time resource management (the means to the objective). In some cases, resource over-

[1] The introduction to this Section and Concepts 1.0 through 1.4 were provided in total by Doug Jensen of The Mitre Corporation. A more detailed discussion of these concepts can be found at http://www.real-time.org.

capacity (e.g., processor performance) may reduce the degree of, or even eliminate the need for, real-time resource management (whether by the programmer or by the computing system).

Rationale:

Sometimes people think of real-time computing solely with respect to the objective of whether or not a system (or a task) meets its timeliness criteria, regardless of the means—e.g., by resource overcapacity and tightly-bounded service latencies, or by explicit timeliness criteria (deadline) driven resource management. Many computing systems include activities which have timeliness criteria at the system levels (e.g., device drivers), or at the application levels (e.g., just-in-time parts delivery deadlines), regardless of whether or not the system uses resource management techniques which are recognized as "real-time." Other times, people think of real-time computing solely with respect to the resource management techniques employed (such as priority-based dispatching and short and tightly upper bounded service latencies in conventional real-time operating systems), and not at all about application tasks having explicit completion time constraints (deadlines) and a scheduling optimization criterion (e.g., meet all deadlines), as discussed below in Section 1.3.1.

1.1 Scheduling and Dispatching

Scheduling is the creation of a schedule: a (partially) ordered list specifying how contending accesses to one or more sequentially reusable resources will be granted. Such resources may be hardware—such as processors, communication paths, storage devices—or they may be software, such as locks and data objects. A schedule is intended to be optimal with respect to some criteria (such as timeliness ones), as discussed below in Section 1.3.1.

In contrast, dispatching is the process of granting access to the currently most eligible contending entity. Eligibility is manifest either by the entity's position in a schedule (the first entity in the schedule has the highest eligibility of all entities in the schedule) or, in the absence of a schedule, by the value of one or more eligibility parameters, such as priority or deadline (the most eligible one has either the highest priority or the earliest deadline, respectively, of all entities ready to access the resource). For simplicity, this paragraph assumes total orders, rather than the more general partial orders.

Eligibility parameters, such as priority and deadline, can be used to create a schedule (in priority or deadline order, respectively). Alternatively, and more commonly in real-time systems and operating systems, they can be used for dispatching from a heap of unscheduled contending entities (highest priority or earliest deadline, respectively)—this is called using a dispatching rule. Popular eligibility parameters, such as priority and deadline, can be employed either for scheduling or for dispatching rules. The earliest deadline first algorithm is often used in both approaches. Both scheduling and dispatching rules always seek to optimize some criterion, as discussed below.

The choice of scheduling vs. dispatching rules should be made on an application-specific basis. Both can be optimal with respect to certain criteria (such as meeting all hard deadlines), but only scheduling can be optimal with respect to certain other criteria (such as maximizing accrued utility). Dispatching rules are greedy in the sense that they consider only the most eligible contending entity. Scheduling is not necessarily greedy (although priority scheduling is) in that it may explicitly take into account the effects on optimality of the sequence of entity accesses to the shared resource. For example, a dispatching rule could choose a task which consumes T units of processor time and yields a utility of eight, and thus choose not to allow two other tasks to execute which together could consume the same T units of processor time but each would yield a utility of five and a sum of ten—a higher accrued utility and utility

per unit time. A scheduling algorithm which seeks to maximize summed utility would make the opposite choice.

Rationale:

The concept of scheduling is widely misunderstood in the practice of real-time computing. Practitioners often speak of scheduling when referring to a dispatching rule, such as priority- or rate- or deadline-based dispatching—but they are not creating schedules (independent of whether that is an appropriate decision for their needs). Moreover, dispatching rules are frequently employed with little or no understanding of what criteria they are and are not optimal for. Another common confusion is to refer to the OS dispatcher—which dispatches from a schedule if a schedule has been provided or from an unscheduled heap otherwise—as a scheduler.

Usually, systems include some entities which are schedulable, and others which are not.

1.2 Schedulable and Non-Schedulable Entities

A computing system usually has a mixture of schedulable and non-schedulable entities. Schedulable entities (e.g., threads, tasks, and processes—in both the application and the system software) are scheduled by the scheduler (which may be part of some system software, such as an operating system, or an off-line person or program). Non-schedulable entities are most often in the system software and can include interrupt handlers, operating system commands, packet-level network communication services, and the operating system's scheduler. Non-schedulable entities can execute continuously, periodically, or in response to events; their timeliness is a system design and implementation responsibility.

Real-time computing practitioners (i.e., real-time OS vendors and users) focus primarily on timeliness in terms of non-schedulable entities (e.g., interrupt response times). Real-time computing principles and theory are focused primarily on timeliness in terms of schedulable entities (e.g., meeting task completion deadlines).

Rationale:

There are two primary reasons to distinguish between schedulable and non-schedulable entities. The first, and more obvious, reason is to illuminate and accommodate the reality that there are both schedulable and non-schedulable entities. The second, and less obvious, reason is that this distinction is a fundamental basis for the wide gulf in the real-time computing field between practitioners and researchers. Real-time computing practitioners (such as users and vendors) usually think predominately about non-schedulable entities—e.g., “hard real-time” in terms of tight upper bounds on OS service latencies and dispatch latencies (OS services are executed when called, and interrupt service routines—at least the “immediate” or “lower” parts—are executed when interrupts occur). Real-time computing researchers usually think predominately with respect to schedulable entities—e.g., “hard real-time” in terms of always meeting all hard deadlines for completing tasks.

1.3 Timeliness Specification

There are two levels of timeliness specification involved in real-time computing: individual and collective. Individual entities—schedulable ones, such as threads, and non-schedulable ones, such as interrupt routines—may each have their own timeliness specification, and sets of entities may have collective timeliness specifications.

Rationale:

Although the real-time computing practitioner community thinks in terms of non-schedulable entities, it does not normally consider that those entities have individual and collective timeliness specifications. Explicitly recognizing these specifications has benefits in both principle and in practice. In principle, both schedulable and non-schedulable entities can then employ the same concepts and terminology; in practice, designers and implementers are then encouraged to think explicitly and systematically about the timeliness of non-schedulable entities.

Schedulable and non-schedulable entities have different styles of timeliness specifications. Timeliness specifications for schedulable entities, which are normally derived from the physical nature of the application environment, are parameters provided to an on-line or off-line scheduler. Timeliness specifications of non-schedulable entities, on the other hand, are more often based on the required properties of the computing system per se (such as network characteristics) or on the generic intention to have minimal latencies. Thus they are not parameters; rather, they are usually designed and implemented into the system regardless of the specific applications.

1.3.1 Schedulable Entity Timeliness Specifications

The first level of timeliness specification for a schedulable entity (such as a thread) is that it may have one or more completion time constraints (e.g., deadlines). The second level of timeliness specification is that the currently runnable collective set of such threads is scheduled according to a policy which seeks to optimize the collective timeliness (e.g., meet all deadlines, minimize mean tardiness, etc.) and to optimize the predictability of that collective timeliness.

Rationale:

In a system with multiple threads with time constraints (e.g., deadlines), requiring that each thread must meet its deadline is equivalent to requiring that all threads must meet their deadlines (a collective scheduling optimization criterion). But meeting all deadlines is only one of many possible scheduling optimization criteria. Two other common criteria include minimizing the number of missed deadlines and minimizing mean tardiness. Any scheduling optimization criterion that can be expressed in terms of each individual thread's timeliness can also be expressed in terms of their collective timeliness, though the converse is not true.

1.3.1.1 Completion Time Constraint

A completion time constraint is a predicate which applies to some portion (frequently all) of a thread's locus of execution; that portion is called the time constraint's scope. The most common example of a completion time constraint is a deadline. Completion time constraints are usually part of the logic of the application and are typically derived from the physical nature of the application.

Rationale:

Completion time constraints are clearly necessary for time constraint based scheduling, and—perhaps not so clearly—they are also sufficient. Thread execution is usually initiated by an event—for example, an external interrupt, precedence constraint satisfaction, resource conflict resolution, or timeout—that makes the thread schedulable according to its eligibility (including its completion time constraint). Of course, a timeliness model that has minimal necessary concepts is not intrinsically more useful than a less minimal one which is more comfortable from the users' preferred view of the system. So, although explicit thread

initiation time constraints are not present in this model, they could be added.

The most common practice is that a thread (or task) is designed to have a single time constraint for the entirety of its execution—in other words, the whole thread has a deadline. This programming style is highly restrictive, because it is often natural from an application standpoint for a given activity, manifest as a thread, to perform a mix of activities—some of which have completion time constraints and others of which do not. Decomposition of that thread into a multiplicity of threads corresponding to separate time constraints—or to no time constraints—is a complicating artifact necessitated by schedulers (usually operating systems) that fail to accommodate arbitrary time constraint scopes for an individual thread.

1.3.1.1.1 Deadline

A deadline is a completion time constraint which specifies that the timeliness of the thread's transit through the deadline 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.

Rationale:

Contrary to popular misconception in the real-time computing domain, a deadline per se (as used throughout fifty years of scheduling theory and practice) has no intrinsic semantics except that timeliness is related in some way to whether or not the deadline is met. Additional semantics are provided by the modifiers “hard” and “soft” in the real-time computing context, and by the scheduling optimization criteria in other (properly, in all) contexts.

1.3.1.1.2 Hard Deadline

A hard deadline is a completion time constraint, such that if the deadline is satisfied—i.e., the thread's execution point reaches the end of the deadline scope before the deadline time occurs—then the time constrained portion of the thread's execution is timely; otherwise, that portion is not timely.

Rationale:

“Hard” adds to “deadline” the semantics of “timely”/“untimely” for the relationship between timeliness and meeting the deadline—but “hard deadline” is often misconstrued to have semantics beyond that. However, the significance of not meeting a hard deadline is specified by the scheduling optimization criterion—most often, all hard deadlines must be met for the system to be operating correctly.

1.3.1.1.3 Soft Deadline

A soft deadline is a completion time constraint, such that if the deadline is satisfied—i.e., the thread's execution point reaches the end of the deadline scope before the deadline time occurs—then the time constrained portion of the thread's execution is more timely; otherwise, that portion is less timely. Thus, a hard deadline is a special case of a soft deadline.

Rationale:

“Soft” adds to “deadline” the semantics of “more timely”/“less timely” for the relationship between timeliness and meeting the deadline—thus, a hard deadline is a special case of a soft deadline, where “more” and “less” timely is binary (timely and untimely). In the soft real-time computing context, and in the general scheduling context, “more” and “less” timely is usually measured in terms of lateness (deadline minus completion time) or tardiness (max{0,lateness}).

1.3.1.1.4 Soft (General) Time Constraint

A soft (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. Soft deadlines are binary-valued special cases of soft time constraints, while hard deadlines are the unit range special case of soft deadlines.

A thread for which there is no relationship between the timing of its execution and its utility to the system is a non-real-time thread.

Rationale:

Deadlines are only one class of completion time constraint: soft deadlines are binary-valued special cases of soft time constraints (and, as described above, hard deadlines are unit range special cases of soft deadlines). Lateness is a linear relationship between completion time and utility. Viewed as a function, unweighted lateness is a line with slope -1 and range { -deadline , infinity }. In general, the utility relationship may be arbitrary. For example, utility might increase in some fashion as completion time increases, reflecting that it is preferable for the thread to complete at a later time (communicating with a satellite has a lower bit error rate as the satellite rises sufficiently above the horizon). Alternatively, utility might remain constant for a range of completion times (as it does for all completion times prior to a hard deadline). Or, utility might decrease in an application- or situation-specific way (often linearly after a soft deadline).

“Non-real-time” is not a first class concept or term here, just as “real-time” is not. The concepts and terms in this document deliberately reflect the fact that “real-timeness” is a continuum, not binary.

1.3.1.2 Collective Timeliness Optimization Criterion

The second level of timeliness specification for schedulable entities in real-time computing is the collective timeliness optimization criterion. It specifies two dimensions of timeliness which are the basis for scheduling multiple threads with time constraints: collective timeliness optimality and the predictability of collective timeliness optimality. (Usually the scheduling policy optimization criterion also contains non-timeliness components, such as relative importance, proportional share, precedence constraints, and resource dependencies.)

1.3.1.2.1 Collective Timeliness Optimality

As used in this context, optimality refers to the merit of a schedule with respect to the timeliness component of the scheduling policy's optimization criterion. Every possible schedule has an optimality value according to this metric. Hard real-time scheduling has only one timeliness component in its optimization criterion—i.e., always meet all hard deadlines—which specifies that only maximum optimality is acceptable. Soft real-time scheduling includes all other possible timeliness components in scheduling optimization criteria; very common examples include minimizing the number of missed deadlines and minimizing the mean tardiness. In general, soft real-time systems are forced by circumstances (e.g., the dynamics of the application and the computing system) to have sub-optimal schedules and thus sub-optimal timeliness for some threads. Determination of which optimality values—and thus which schedules—are acceptable is application-specific. In real-time computing, predictability of the optimality values is generally as important as the values per se.

Rationale:

A set of ready to run threads with individual time constraints could in general be scheduled to run in a multiplicity of different sequences. The application(s) and system almost always prefer some possible schedules over others; the basis for this preference is called the scheduling optimization criterion. Collective (and thus individual) thread timeliness is one factor in this criterion. A scheduling algorithm is employed which uses this criterion. Scheduling in general—and almost all interesting scheduling problems in particular—are NP-hard (i.e., require time which increases exponentially with the number of scheduled entities); thus, optimum schedules are the exception rather than the rule. Hard real-time scheduling is one of those fortunate exceptions: simple algorithms exist that—given their prerequisite conditions are satisfied—produce schedules which satisfy the timeliness factor of the optimality criterion to meet all hard deadlines (any such schedule is acceptable). Soft real-time scheduling is required when the hard real-time prerequisites cannot be satisfied, or when soft deadlines or general time constraints are more appropriate. Soft real-time scheduling optimization criteria and scheduling algorithms are more complex than the hard real-time ones, but are applicable to a much broader range of timeliness requirements and application/system conditions.

1.3.1.2.2 Predictability

A property is predictable to the degree that it is known in advance. One endpoint of the predictability scale is determinism, in the sense that the property is known exactly in advance. The other endpoint is maximum entropy, in the sense that nothing at all is known in advance about the property. In stochastic systems (which include hard real-time ones as a special case), one way to measure predictability is the coefficient of variation C_v: the maximum predictability endpoint is the deterministic distribution (whose $C_v=0$), and the minimum endpoint is the extreme mixture of exponentials distribution (whose $C_v=8$).

Rationale:

Predictability of timeliness is the most fundamental property of a real-time system. And yet, predictability is among the most misunderstood concepts in real-time computing practice and even real-time computing theory. The two most frequently encountered manifestations of this misunderstanding are use of the terms “deterministic” and “predictable” interchangeably, and misuse of each of those terms individually. It is not necessary to involve physics or probability theory to arrive at an understanding of these concepts and terms which is sufficiently accurate and useful for real-time computing.

1.3.1.2.3 Predictability of Timeliness Optimality

The most important timeliness predictability is normally the predictability of collective timeliness optimality, rather than predictability of each individual thread's completion with respect to its own time constraint. In other words, it is the performance of the system as a whole (e.g., meet all deadlines, minimize the mummer of missed deadlines, etc.) which is most important. In principle, the predictability of any particular individual entity can be specified as part of the collective timeliness optimality criterion (e.g., meet thread T_a's and T_b's deadlines deterministically, and minimize the mean tardiness of all other threads' deadlines). In practice, it is simpler for the optimization criterion and scheduling algorithm to distinguish between hard and soft time constraints, or take relative importance into account, than it is to distinguish individual entities.

Rationale:

It is often natural to think about the predictability of the timeliness of one or more specific entities. For example, one might want the probability of meeting a deadline to be different for different threads,

perhaps depending on the lateness of certain preceding threads. But scheduling theory is primarily focused on the more tractable goal of scheduling the execution of all entities such that their collective timeliness is optimal according to some criterion (even so, most of the interesting cases are still intractable). Nonetheless, collective timeliness of the whole system, though often complex to specify, is the ideal measure. Heuristics are necessary in all but the simplest real-time systems.

1.3.1.2.4 Orthogonality of Collective Timeliness Optimality and Predictability

The two dimensions of collective timeliness—optimality and predictability of optimality—are orthogonal and generally must be traded off against one another on an application-specific basis. For example, it may be necessary to choose between one schedule which results in better optimality (e.g., lower mean tardiness) with a worse (higher) coefficient of variation and another schedule which results in worse optimality (e.g., higher mean tardiness) with a better (lower) coefficient of variation. Hard real-time is a special case where this tradeoff is not necessary.

Rationale:

This is self-evident in theory and demonstrably true in practice.

1.3.2 Non-Schedulable Entity Timeliness Specifications

The timeliness specifications for non-schedulable entities correspond exactly to those for schedulable entities. Hard and soft upper bounds replace hard and soft deadlines; collective timeliness is unchanged, except for being a design and implementation time responsibility; and predictability of timeliness is unchanged.

Rationale:

The timeliness specifications of non-schedulable entities are important, both because such entities perform functions which may be just as important as those of schedulable entities, and because they consume time and resources which scheduling cannot take into account. Upper bounds provide a means for assessing the timeliness of non-schedulable entities, and for accounting for their impact on the timeliness of schedulable entities. In real-time computing practice, upper bounds on latencies of non-schedulable entities (particularly interrupt response times and OS service times) are the primary measure of timeliness (whether that should be so or not). However, practice does not normally extend to recognizing that non-schedulable entities may have timeliness properties analogous to those of schedulable entities.

1.3.2.1 Non-Schedulable Entity Completion Time Constraint

The completion time constraints of non-schedulable entities are: hard and soft upper bounds on their execution latencies (durations), which correspond to hard and soft deadlines of schedulable entities; and general time constraints on their latencies, which correspond to general time constraints on schedulable entities.

1.3.2.1.1 Upper Bound

An upper bound in this context is a completion time constraint which specifies that the timeliness of the non-schedulable entity's transit through the upper bound's scope depends on whether the entity's

execution point reaches the end of the scope before the upper bound time has occurred, in which case the upper bound is satisfied.

1.3.2.1.2 Hard Upper Bound

A hard upper bound in this context is a completion time constraint, such that if the upper bound is satisfied—i.e., the non-schedulable entity's execution point reaches the end of the upper bound scope before the upper bound time occurs—then the time constrained portion of the entity's execution is timely; otherwise, that portion is not timely.

1.3.2.1.3 Soft Upper Bound

A soft upper bound in this context is a completion time constraint, such that if the upper bound is satisfied—i.e., the non-schedulable entity's execution point reaches the end of the upper bound scope before the upper bound time occurs—then the time constrained portion of the entity's execution is more timely; otherwise, that portion is less timely. Thus, a hard upper bound is a special case of a soft upper bound.

1.3.2.1.4 Non-Schedulable Entity Soft (General) Time Constraint

A soft (general) time constraint is any relationship between the time when the non-schedulable entity's execution point reaches the end of that time constraint's scope, and the utility to the system of when it does so. Soft upper bounds are binary-valued special cases of soft time constraints, and hard upper bounds are the unit range special case of soft upper bounds.

A non-schedulable entity for which there is no relationship between the timing of its completion and its utility to the system is a non-real-time entity.

1.3.2.2 Collective Non-Schedulable Entity Timeliness Optimization Criterion

Collective timeliness optimality of non-schedulable entities is analogous to that of schedulable entities, except that seeking the optimality is the responsibility of the system/application design and implementation process. Non-schedulable entity collective timeliness optimality criteria can be employed which are analogous to those for schedulable entities (meet all hard upper bounds, minimize the mean tardiness according to relative importance, etc).

1.3.2.3 Predictability of Non-Schedulable Entity Collective Timeliness Optimality

Predictability of timeliness optimality for non-schedulable entities is analogous to that for schedulable entities.

1.3.3 Hard Real-time

Hard real-time is the case where: for the schedulable entities, some time constraints are hard deadlines, and the timeliness component of the scheduling optimization criterion is to always meet all hard deadlines (additional components may apply to any soft time constraints); for the non-schedulable entities, some upper bounds are hard, and the system has been designed and implemented so that all hard upper bounds

are always satisfied (other non-schedulable entities may have soft upper bounds). Thus, the feasible schedules (with respect to those schedulable entity time constraints) are always optimal, and the predictability of that optimality is maximum (deterministic).

Rationale:

Hard real-time is one of the basic concepts in real-time computing which is almost universally misunderstood in practice. While practitioners typically think primarily in terms of non-schedulable entities, real-time computing theoreticians typically think exclusively in terms of schedulable entities. An effective definition must not only be in terms of collective completion timeliness, but also in terms of both schedulable and non-schedulable entities.

1.3.4 Soft Real-time

Soft real-time represents all cases which are not hard real-time (soft real-time is the general case, of which hard real-time is a special case). Time constraints are soft (which may include the hard deadline special case), such as the classical lateness function. Any scheduling optimization criteria may be used (including the hard real-time special case), such as minimizing the number of missed deadlines, or minimizing mean tardiness, or maximizing the accrued utility. Predictability of schedule optimality (and thus thread timeliness) is generally sub-optimal, but may be deterministic (including but not limited to the special hard real-time case). Upper bounds are soft, and predictability of non-schedulable entity timeliness is generally sub-optimal.

Rationale:

Soft real-time is even less well-defined than hard real-time, because even the real-time computing theoreticians tend to leave it at the tautology “not hard real-time.” Soft real-time corresponds directly to a precisely defined corresponding concept in scheduling theory.

1.4 Real-time Resource Management

Resource management is real-time to the degree that it explicitly employs the time constraints (e.g., deadlines) of the schedulable entities (such as threads). Real-time resource management can be done by the programmers a priori or by the computing system at runtime.

Rationale:

It is useful to have a qualitative basis for considering the degree to which resource management is real-time—one that is consistent with the real-time computing concepts and terms. But it does not appear useful to have a quantitative measure.

Terms

Application binding time: The time at which a decision as to what features will be present in the deployed environment is made. This includes whole classes, methods, and fields. Application binding time is usually the latest time at which symbolic resolution can possibly occur. The EmbeddedJava™ technologies allow a form of ahead-of-time application binding, where developers can specify which features will be required by their application and which will not be required (this distinction can potentially be made before the application code exists).

Asynchronous event/Synchronous event: A synchronous event is one that is triggered by some action of a flow of control, and therefore its timing is predictably tied to that flow. An asynchronous event is one

that is triggered by some action that is not part of a flow of control, and therefore its timing is unpredictable with respect to that flow of control.

Atomic: Refers to an operation that is never interrupted or left in an incomplete state under any circumstances [Javasoft].

Component: "Black box" components are sufficiently characterized by their specification and are delivered in bytecode form. For real-time systems, "sufficiently characterized" includes timing information and perhaps the rate of garbage production. For embedded systems, the resource footprint is also an issue.

Garbage: The heap is a region of memory within which objects of temporary utility (heap objects) are allocated. Heap objects that are referenced from the programmer-declared local variables of any Java thread and heap objects referenced from specially designated native memory locations identified as root pointers are called “live objects”. All other objects are called “garbage”.

Garbage collection (GC): The process that automatically identifies dynamically allocated objects that are no longer reachable (garbage) and reclaims the space occupied by those objects. Garbage collection consists of both garbage detection and garbage reclamation.

Interpreter: The module that alternately decodes and executes every statement in some body of code. The Java interpreter decodes and executes Java bytecode [Javasoft].

Negotiating component (NC): A “place holder” name for a type of component that implements a set of special interfaces to communicate with the Java platform about its resource requirements.

Process: A virtual address space containing one or more threads [Javasoft].

Real-time component: see Negotiating Component.

Real-time garbage collection: Garbage collection characterized by having bounded system pause time, a guaranteed rate of memory reclamation, and a bounded allocation time. The system pause time is the time required to preempt garbage collection activities when a higher priority thread becomes ready to run. The rate of memory reclamation may depend on the rate of garbage creation, which is determined by the system workload. The bound on allocation time is the maximum amount of time required to allocate new objects assuming the rate of new memory allocation does not exceed the rate of memory reclamation. The bound on allocation time may be a function of the size of the allocation.

Real-time Java infrastructure: Consists of the Java Platform with Real-time extensions and tools to provide real-time capabilities to the Java application.

RTJ: Acronym used to define the functionality provided by the Real-time extensions to the Java Platform.

Runtime system: The software environment in which programs compiled for the Java Virtual Machine (JVM) can run. The runtime system includes an implementation of the JVM, which may be a Java interpreter, and all the code necessary to load Java programs, dynamically link native methods, manage memory, and handle exceptions. [Javasoft].

Symbolic resolution time: The time at which symbolic references within a program are resolved. This is analogous to link time in a C++ program. In most implementations of the Java platform, this symbolic resolution is done at runtime. Dynamic classloading requires symbolic resolution at runtime. Some systems allow Java classes to be efficiently stored in ROM, which can move some or all symbolic resolution to part of the build process. Static ("ahead of time") compilers often move symbolic resolution to build time—except, of course, for dynamically loaded code.

Thread: A schedulable entity that contends for a resource and is scheduled to acquire it [Jensen].

Sources

[Javasoft] Sun Microsystems, Inc. provides an on-line glossary of Java related terms. See http://www.Javasoft.com/docs/glossary.print.htm.

[Jensen] Doug Jensen, of The Mitre Corporation, provides an excellent source of explanatory material on real-time concepts at http://www.realtime-os.com/rt-java_glossary/transit/rt-java_glossary.htm.

Section 2 — Java Traits

The Requirements Group considers these Java traits to provide a basis for the real-time requirements and motivation for Java's use by the real-time community:

- Java's higher level of abstraction allows for increased programmer productivity (although recognizing that the tradeoff is runtime efficiency).
- Java is relatively easier to master than C++.
- Java is relatively secure, keeping software components (including the JVM itself) protected from one another.
- Java supports dynamic loading of new classes.
- Java is highly dynamic, supporting object and thread creation at runtime.
- Java is designed to support component integration and reuse.
- The Java technologies have been developed with careful consideration, erring on the conservative side using concepts and techniques that have been scrutinized by the community.
- The Java programming language and Java platforms support application portability.
- The Java technologies support distributed applications.
- Java provides well-defined execution semantics.

Section 3 — Guiding Principles

The following three guidelines were used by the Requirements Group to develop the cross-disciplinary requirements for real-time functionality that are expected to be needed by real-time applications written in the Java programming language and executed on various Java platforms.

1. The design of RTJ[2] may involve compromises that improve ease of use at the cost of less than optimal efficiency or performance.

[2] As defined in the Terms section, "RTJ" is used to identify the sum of the functionality and services that would be provided through real-time extensions for the Java platform.

2. RTJ should support the creation of software with useful lifetimes that span multiple decades, maybe even centuries.

3. RTJ requirements are intended both to be pragmatic, by taking into account current real-time practice, and visionary, by providing a roadmap and direction to advance the state of the art.

Discussion:

The current predominant practice for development of real-time software uses the C programming language in combination with real-time operating systems (either commercial or proprietary). The Requirements Group recognizes the following real-time operating system services as key: (1) support for fixed priority, round-robin scheduling; (2) mutual exclusion locking with some form of priority inversion avoidance protocol (such as priority inheritance or priority ceiling); (3) inter-task synchronization (such as is provided by semaphores); (4) an ability to write interrupt handlers and device drivers; (5) an ability to manage interrupts (enabling, disabling, prioritizing); and (6) an ability to timeout or otherwise abort a running task.

In the real-time operating system marketplace, key technical product differentiators include: (1) small memory footprint, (2) fast interrupt response latency, (3) breadth of general-purpose operating system services, and (4) host-target cross development tools. Today's commercially successful real-time operating systems offer configurations that range in size from less than 10 Kbytes to several megabytes; offer response latencies measured in tens of microseconds; include libraries for graphics applications, network sockets, and file and device I/O; and provide full-featured symbolic debuggers and execution performance monitoring tools.

As the domain and sophistication of real-time systems expand, there is considerable opportunity to advance the state of the art. Opportunities that are recognized and valued by the Requirements Group include: (1) automated support for execution time and schedulability analysis and (2) support to allow independent real-time software components to determine their resource needs and to negotiate for the resources they require. In general, the vision of advancing the state of the art is to enable the development and widespread deployment of portable real-time software components.

Consistent with the current state of practice, the base real-time Java specification must provide mechanisms to allow the following minimal set of real-time operating system capabilities:

1. Support for fixed priority, round-robin scheduling.
2. Mutual exclusion locking with some form of priority inversion avoidance protocol.
3. Inter-task synchronization.
4. An ability to write interrupt handlers and device drivers.
5. An ability to associate a segment of code with a hardware interrupt (for example, by hooking into the vector table or wrapping the code segment in an interrupt service thread).
6. An ability to timeout or otherwise abort a running task.

To the extent reasonably possible, real-time profiles (if not the base set of requirements) should address the special needs of particular real-time constituencies as characterized by: (1) small memory footprint, (2) fast interrupt response latency, (3) breadth of general-purpose operating system services, and (4) cross development tools. Future profiles and/or evolution of the base or existing profiles should support advancement of the state of the art, as described above.

Section 4 — Core Functionality and Profiles

Because of the wide variety of applications that need to be served by these requirements, they should be partitioned into a small set of profiles built on top of a common RTJ core. This partitioning should result in the smallest set of profiles possible to meet the varying needs of users. Specifications should provide a framework for these profiles. The RTJ core consists of real-time functionality distinct from the underlying Java specification. This core should consist of real-time functionality that addresses the needs of common real-time applications. One way to describe a possible level of core functionality is to consider the intersection of the capabilities provided by widely available real-time operating systems.

Profiles should contain a set of related functionality that is useful to be added to a system as a unit. Profiles may augment or restrict the base core as necessary. It is expected that in some cases profiles will be mutually exclusive. Implementations of the specification should choose which profiles to implement. All implementations must implement the core except as defined by profiles that restrict the core. The specification must allow implementations that implement only those parts of the core that are required in the presence of one or more restrictive profiles. For an implementation to be described as supporting a given profile, it must make all and only the functionality of that profile available at application binding time. Examples of application binding time include but are not limited to the point where classes are loaded dynamically, or when an application is placed in ROM. [Editors' note: Consensus was not reached regarding restrictive profiles.]

The core of real-time functionality must be established for the profiles to augment and or restrict.

Some examples of possible profiles:

1. Safety critical,
2. High availability and fault tolerant,
3. Low latency,
4. Deadline-based scheduling, or priority, or round-robin, or none,
5. No dynamic loading,
6. Bare bones, and
7. Distributed real-time.

Section 5 — Core Requirements

[Editors' note: Unless explicitly stated otherwise, the Requirements Group reached consensus on each of the following requirements and derived requirements.]

Core Requirement 1: The specification must include a framework for the lookup and discovery of available profiles. For example, existence of a profile; enumeration of profiles; availability of profiles for loading; and, potentially, versioning.

Core Requirement 2: Any garbage collection that is provided shall have a bounded preemption latency. The preemption latency is the time required to preempt garbage collection activities when a higher priority thread becomes ready to run. The specification must clearly define the restrictions that are imposed by preemption. (For example: Can preempting tasks allocate memory? Can they refer to existing

objects?)

Rationale:

Garbage collection activity will need to be preempted, just like other activities in a priority-preemptive scheduling environment. In order to produce a valid real-time schedule, the preemption latency associated with garbage collection must be bounded.

Timely preemption of a garbage collector may put the heap under its management in an inconsistent state. Thus, it may not be possible for a preempting real-time thread to act upon that heap.

Dynamic memory de-allocation in Java is implicit; thus, if dynamic memory management is required for a given application, garbage collection is an appropriate technique. However, many real-time systems, especially hard real-time systems, do not require dynamic memory management, and have no need for its space and time overhead. It is possible to create real-time Java applications that do not require any dynamic memory management. Further, it is possible to create low-latency real-time Java applications in which dynamic memory management is present, but is not needed to service high-priority, low-latency events. Bounded preemption time for the garbage collector is thus required.

Derived Core Requirements for Core Requirement 2

DCR 2.1: For the functionality (language and libraries) that are expected to be used by real-time threads, a tight upper bound on the memory resources required by that functionality must be quantifiable prior to runtime. (For example, how much memory is required to represent objects of a particular type? How much memory will be allocated by invoking a particular method?)

Rationale:

This requirement is traceable to Goal 8 below (real-time Java should allow resource reservations and should enforce resource budgets). This requirement is particularly important to those designers who wish to pre-allocate memory and forego garbage collection for a particular application.

DCR 2.2: The specification must identify the set of functionality that may be used by real-time threads. Possible restrictions on these threads are not necessarily the result of only memory allocation (e.g., pointer assignments that change the liveness of an object).

Rationale:

The behavior of real-time threads may need to be restricted so as not to interfere with the preempted system state. For example, timely preemption of a garbage collector may put the heap under its management in an inconsistent state. Thus, it may not be possible for a preempting real-time thread to act upon that heap.

DCR 2.3: RTJ must not require bounds on when or whether an object is finalized or reclaimed.

Rationale:

Such bounds would be conflict with the Java Language Specification.

DCR 2.4: Within RTJ, the GC overhead, if any, on the application must be quantified.

Rationale:

Garbage collection activity must be scheduled like any other activity in a real-time system. If the GC overhead is not quantifiable, a valid real-time schedule cannot be constructed.

Core Requirement 3: The RTJ specification must define the relationships among real-time Java threads at the same level of detail as is currently available in existing standards documents. An example of the

minimal requirements for specification would be POSIX 1003.1B. Examples of these relationships include thread scheduling, wait queue ordering, and priority inversion avoidance policies.

Core Requirement 4: The RTJ specification must include APIs to allow communication and synchronization between Java and non-Java tasks.

Core Requirement 5: The RTJ specification must include handling of both internal and external asynchronous events. The model must support a mechanism for executing Java code in response to such events. This mechanism should fit in well with existing Java mechanisms (such as `wait/notify`). [Editors' note: There was no agreement whether general asynchronous events that target individual threads are required.]

Core Requirement 6: The RTJ specification must include some form of asynchronous thread termination. This asynchronous thread termination must have the following properties:

1. By default a thread cannot be aborted.
2. The target code determines where it can be aborted.
3. When a thread is aborted:
    a. all locks are released, and
    b. `finally` clauses execute on the way out when the thread is being aborted.
4. No facility for aborting uncooperative code need be provided. The termination shall be deferred if the code currently executing has not indicated a willingness to be terminated.
5. Mechanisms must be provided that allow the programmer to insure data integrity.

Core Requirement 7: The RTJ core must provide mechanisms for enforcing mutual exclusion without blocking. This requirement does not imply that a real-time thread should be allowed to disable/enable interrupts to achieve these results.

The specification must require that the mechanisms do not allow a non-real-time thread to gain complete control of the machine. Specifically, the scheduler will continue to dispatch threads and interrupt handlers, other than the one possibly attached to the thread using the non-blocking mutex, which will continue to execute. The specification should take care to minimize the risks to the system integrity, possibly by integrating with Java's existing security manager.

Rationale:

The initial thought for "mutual exclusion without blocking" was a mechanism to construct atomic sequences of Java instructions. This mechanism cannot do anything that could not be done with any locking mechanism, but this gives the language a way to expose any efficient mechanisms for atomic sequences that might be supported by the underlying platform.

The obvious mechanism is masking interrupts. Provided that the atomic interval is brief, masking interrupts is a standard way for operating system code to guarantee that an instruction sequence will not be interrupted.

Masking interrupts has three problems:

1. Masking interrupts is not sufficient if multiple processors are executing Java threads.
2. Masking interrupt capability is not available unless Java is running in system state.
3. Masking interrupts affects the entire machine. Non-Java interrupts are also masked, and there are no requirements to limit the duration of the atomic section.

The requirement rules out masking interrupts since the dispatcher is required to continue to switch threads and to service interrupts. This removes the above problems, but leaves a puzzle.

A possible solution is to provide a "conditional acquire lock" function that will never block, but will only return success if the resource is not locked and it acquires the lock. That seems to meet the requirement, but that solution was never raised as a possibility during the Requirements Group's meetings. Conditional lock would lead toward the family of two-phase commit-like procedures instead of the simple start-atomic/end-atomic model proposed.

Possibly, the non-blocking mutual exclusion requirement can be met by pretending not to block.

The system could use blocking mutual exclusion with a global lock shared by all threads. True, a process might block when it tried to start the mutual exclusion section, but it would behave just the same if another thread started a non-blocking mutual exclusion at just that moment.

A long atomic section—or one that could include garbage collection or finalizer execution, exceptions, or other hard-to-predict code—could be a serious problem for the real-time characteristics of the entire application. It would be possible to ask that atomic sections be restricted in what they can do and how long they can run. However, such a restriction would probably require a JVM change. The length restriction would lead to a requirement that the code in the atomic section be "analyzable" (e.g., an automated tool that cannot set a ceiling on the number of instructions that will be executed in the atomic section would be illegal).

[Editors' note: For historical accuracy, it should be noted that the following requirement was removed by consensus: The RTJ specification must provide mechanisms for ensuring data integrity in the presence of asynchronous event handling which may result in changes in the control sequence (such as provided by the PERC atomic statement).]

Core Requirement 8: The RTJ specification must provide a mechanism to allow code to query whether it is running under a real-time Java thread or a non-real-time Java thread.

Core Requirement 9: The RTJ specification must define the relationships that exist between real-time Java and non-real-time Java threads. It must provide mechanisms for communication and sharing of information between real-time Java and non-real-time Java threads.

Derived Core Requirements for Core Requirement 9

DCR 9.1: Traditional Java software must run as non-real-time tasks.

[Editors' note: For historical accuracy, it should be noted that the following derived requirement was removed by consensus: Facilities must be provided to allow sharing of information between non-real-time and real-time tasks.]

DCR 9.2: The sharing and communications protocols must have known tight upper bounds or some other form of predictability on blocking delays (see section 1.2.1.2.2. above for the definition of "predictable").

DCR 9.3: The relationships between RTJ threads and the other three possible types of threads (non-real-time Java, non-Java real-time, and non-Java non-real-time) need to be defined.

These relationships include:

- Priorities and other scheduling relationships,
- Sharing resources (memory, devices),
- Other processes,
- Synchronization,

- Budgets (memory, CPU, other resources), and
- Protections and privileges.

[Editors' note: The definition of the relationships between the threads may be imprecise, recognizing that it is not possible to standardize the behavior of Java non-real-time tasks and the behavior of non-Java tasks, both of which are outside the control of the Real-time Java specification effort.]

Rationale:

Systems that run real-time Java threads may also run conventional Java threads and threads that are not Java at all. In a typical embedded system, all the Java threads could be real-time, but there will most likely be non-Java threads running on the same hardware. A larger-scale system, moreover, is likely to add non-real-time Java threads to the mix.

That means that there are two sets of relationships: one with the Java threads that do not obey the real-time rules, and one with the non-Java threads (whether real-time or not).

Relationships with non-real-time Java threads are similar to the relationships with real-time Java threads (see Core Requirement 3 above). Non-real-time threads cannot be expected to obey any special rules and guidelines required of real-time threads, and they need to operate "normally" in the presence of real-time threads. Consequently, real-time threads have to communicate with non-real-time threads using currently supported mechanisms, and the currently supported mechanisms can only have visible new behavior at the real-time end.

The specification must answer such questions as: Can a non-real-time thread have higher priority than a real-time thread? Do all the inter-thread communication and synchronization mechanisms currently defined work between real-time and non-real-time threads? If a real-time and a non-real-time thread meet at an explicit or implicit lock, what happens? What if a thread terminates while it is holding a lock?

Careful specification for the interactions between real-time threads is required in Core Requirement 4. Core Requirement 9 extends those requirements to all the Java mechanisms that: (1) could be used by a real-time thread, and (2) are not already carefully specified because of Core Requirement 3.

There is the option to say that a well-behaved real-time thread will not do *X*. In that case, the behavior of *X* need not be fully characterized. If, however, a non-real-time thread can do *Y* to a real-time thread without its consent, the characteristics of *Y* must be fully specified.

The standard Java platform does not provide a suitable mechanism for communication between real-time and non-real-time threads. The mechanism preferred by the Requirements Group must allow the real-time thread to control whether it will block or not. Furthermore, the mechanism should be simple and efficient.

It is not required that the mechanism be simple message queues, but they seem like the best fit.

Section 6 — Goals and Derived Requirements

The Requirements Group defined thirteen goals that should be addressed, where appropriate, in the base real-time Java specification, future profiles, or a future base specification. Some of the goals are accompanied by derived requirements. Many of these requirements are intended for future use. The Requirements Group purposely added requirements that would create discussion and generate future innovation.

Goal 1: RTJ should allow any desired degree of real-time resource management for the purpose of the system operating in real-time to any desired degree (e.g., hard real-time, and soft real-time with any time

constraints, collective timeliness optimization criteria, and optimality/predictability tradeoffs).

Goal 2: Support for RTJ specification should be possible on any implementation of the complete Java programming language.

Derived Requirements for Goal 2

DR 2.1: RTJ programming techniques should scale to large or small-memory systems, to fast or slow computers, to single CPU architectures and to SMP machines.

DR 2.2: RTJ should support the creation of both small, simple systems and large, complex systems (possibly using different "profiles").

DR 2.3: Standard subsets of RTJ and RTJVM specifications should be created as necessary to support improved efficiency and/or reliability for particular specialized domains.

Goal 3: Subject to resource availability and performance characteristics, it should be possible to write RTJ programs and components that are fully portable regardless of the underlying platform.

Derived Requirements for Goal 3

DR 3.1: Minimal human intervention should be required when the software is "ported" to new hardware platforms or combined with new software components.

DR 3.2: RTJ should abstract operating system and hardware dependencies.

DR 3.3: RTJ must support standard Java semantics.

DR 3.4: The RTJ technologies should maximize the use of non-RTJ technologies (e.g., development tools and libraries).

DR 3.5: The RTJ API must be well-defined with guarantees on all language features.

Goal 4: RTJ should support workloads comprised of the combination of real-time tasks and non-real-time tasks.

Goal 5: RTJ should allow real-time application developers to separate concerns between negotiating components.

Goal 6: RTJ should allow real-time application developers to automate resource requirements analysis either at runtime or off-line.

Goal 7: RTJ should allow real-time application developers to write real-time constraints into their software. [Editors' note: Consensus was achieved with serious reservations regarding the ramifications of the clauses below.]

Derived Requirements for Goal 7

DR 7.1: RTJ should provide application developers with the option of using conservative or aggressive resource allocation. [*Open — no consensus*]

DR 7.2: The same RTJVM should support combined workloads in which some activities budget aggressively and other conservatively. [*Open — no consensus*]

DR 7.3: RTJ infrastructure should allow negotiating components to take responsibility for assessing and managing risks associated with resource budgeting and contention.

DR 7.4: RTJ should allow application developers to specify real-time requirements without understanding "global concerns". For example, a negotiating component should speak in terms of deadlines and periods rather than priorities.

DR 7.5: RTJ must provide a mechanism to discover the relationship between available priorities for Java threads and the set of all priorities available in the system. In addition, a mechanism must be provided to allow the relationships between Java priorities and system priorities to be determined. (This does not have to be a single call.)

Goal 8: RTJ should allow resource reservations and should enforce resource budgets. The following resources should be budgeted: CPU time, memory, and memory allocation rate.

Derived Requirements for Goal 8

DR 8.1: RTJ must:

- At least support strict priority-based scheduling, queuing, and lock contention. This support should apply to existing language features as well.
- At least support some kind of priority "boosting" (either priority inheritance or priority ceilings). This support should apply to existing language features as well.
- Support dynamic priority changes.
- Support the ability to propagate a local priority and changes to remote servers—not just in support of RMI but also in support of user-written communication mechanisms.
- Support the ability to defer asynchronous suspension or disruption when manipulating a data structure.
- Support the ability to build deadline-based scheduler on top.
- Support the ability to query to find out the underlying resource availability (non-Java) and handle asynchronous changes to it.

DR 8.2: Language and libraries must be clearly understood in terms of memory usage.

DR 8.3: RTJ shall provide support for a guaranteed allocation rate.

DR 8.4: RTJ must not require bounds on when an object is finalized or reclaimed.

DR 8.5: RTJ should provide for specifying memory.

DR 8.6: The priority mechanism must take into consideration the existing security protocols related to setting priorities to high levels.

Goal 9: RTJ should support the use of components as "black boxes"; including such use on the same thread. [*Open — no consensus*]

Derived Requirements for Goal 9

DR 9.1: RTJ should support dynamic loading and integration of negotiating components.

DR 9.2: RTJ should support a mechanism for negotiating components whereby the behavior of critical sections of code is locally analyzable.

DR 9.3: RTJ should support the ability to enforce (with notification, event handling and accounting) space/time limits, in a scoped manner, from the outside (on "standard" Java features as well).

DR 9.4: In a real-time context, existing Java features should "work right", including `synchronized` (bounded priority inversion) and `wait/notify` (priority queuing).

Discussion:

Any thread created in "scope" of user-defined policy manager is under control of that policy. Pre-existing thread class priorities might be mapped to real-time priorities.

Goal 10: RTJ must provide real-time garbage collection when garbage collection is necessary. GC implementation information must be visible to the RTJ application (see above for the definition of the term “real-time garbage collection”).

Derived Requirements for Goal 10

DR 10.1: RTJ defines “garbage”.

DR 10.2: RTJ should provide “hint handling” information regarding the GC (e.g., accurate vs. conservative? Does it defragment?). [*Open — no consensus*]

DR 10.3: RTJ must not restrict nor specify the garbage collection technique; rather, it should be capable of supporting all appropriate techniques for real-time GC.

DR 10.5: The GC must make forward progress at some rate. The rate must be “queryable” and configurable.

DR 10.6: Within RTJ, the GC overhead, if any, on the application must be quantified.

Goal 11: RTJ should support straightforward and reliable integration of independently developed software components (including changing hardware).

Goal 12: RTJ should be specified in sufficient detail to support (and with particular consideration for) targeting by other languages, such as Ada.

Goal 13: RTJ should be implementable on operating systems that support real-time behavior.