NISTIR XXXX - WORKING DRAFT

October 28, 1998

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

Editor

U.S. DEPARTMENT OF COMMERCE

Technology Administration

Information Technology Laboratory

National Institute of Standards and Technology

Gaithersburg, MD 20899-0001

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™ 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.

Aonix

Apogee Software

Bay Networks / Bay Architecture Lab

Commotion Technology, Inc.

Enea OSE Systems

Florida State University, Department of Computer Science

Hewlett-Packard Company

Honeywell

IBM

Insignia Solutions, Inc.

Integrated Systems, Inc.

Intermetrics, Inc. (An AverStar Company)

Mantha Software, Inc.

Microsoft

Microware Systems Corporation

Motorola, Inc.

NewMonics, Inc.

NIST/Information Technology Laboratory

Nokia Research

NSI Com

Oberon Microsystems

OMRON Corporation

Plum Hall, Inc.

Plum Hall Europe, Ltd.

QNX Software Systems, Ltd.

Rockwell Collins

Schneider Automation

Siemens AG, A&D

Silverhill Systems, Inc.

SoftPLC Corporation

Sun Microsystems, INC.

TeleMedia Devices, Inc.

The MITRE Corporation

The Open Group

Wind River Systems, Inc.

Table of Contents

Section 1 - Terms and Concepts

Real-time Concepts

Real-time Computing has two aspects: ‘operating in real-time’ and real-time resource management. A system (at any level of abstraction -- e.g., application, computer system, operating system, Java platform, etc.) operates in real-time to the degree that it satisfies the system's time constraints acceptably -- i.e., with acceptable optimality and predictability. To that end, resource management is real-time to the degree that it explicitly takes those time constraints into account. [Abstracted from Jensen]

Timeliness can be defined by three aspects: (1) any time constraints (e.g., deadlines) on individual activities (an activity is a portion of a thread's locus of execution); (2) the collective timeliness optimization criterion used for scheduling threads to meet their activity’s time constraints (e.g., meet all hard deadlines, minimize the number of missed deadlines according to their relative importance, minimizise mean tardiness, etc.), (3) the trade-off between the collective timeliness optimality and the predictability of that collective timeliness optimality value (e.g., mean tardiness or number of missed deadlines may be higher but their variance is lower, or vice versa). [Abstracted from Jensen]

Predictability is the degree to which something can be known a' priori. It implies the degree to which the future execution (system and application) context of that something can be predicted. Determinism and maximum entropy are the end-points on the scale of predictability, and coefficient of variation is an example of a predictability metric. [Abstracted from Jensen]

Hard DeadlineA hard deadline is an activity time constraint which requires that the activity complete by the deadline time in order to be considered timely; completion after the deadline is considered untimely (i.e., there is no utility in completing the activity). [Abstracted from Jensen]

Hard real-time Computing is the fulfillment of the timeliness property that: 1) the time constraints of interest are all hard deadlines; 2) the collective timeliness optimization criterion is to always meet all hard deadlines (optimality and predictability are both maximal). [Abstracted from Jensen]

A soft time constraint is one which is not hard -- i.e., an arbitrary function relating the utility of completing the activity to its completion time. [Abstracted from Jensen]

Soft Real-time Computing is a fulfillment of the timeliness property that: 1) the time constraints are soft; the collective timeliness optimization criterion is whatever is appropriate (e.g., certain activities can miss every nth deadline or can be x% late, minimize the number of missed deadlines according to their importance, minimize mean tardiness); 3) the trade-off between collective timeliness optimality and predictability is whatever is appropriate (e.g., less optimal but more predictable, or more optimal but less predictable). [Abstracted from Jensen]

Real-time Scheduling

Real-Time Java (RTJ) functionality must support the ability to set and modify the priorities for threads.

Priorities are meaningful in the sense that a thread with a higher priority always gets preferential CPU access compared to a thread with a lower priority. Wait queues associated with semaphores and monitors are ordered by priority.

RTJ should also support the ability to specify deadlines for particular real-time threads. If a workload consists of some threads that have priorities and other threads that have deadlines, the system works first to satisfy deadlines and second to schedule whatever CPU time remains giving preference to those threads that have higher priority. Insofar as sorting order for semaphore and monitor queues is concerned, threads with deadlines are always sorted ahead of threads with priorities.

RTJ uses priorities and deadlines to specify the urgency of particular threads. RTJ should also provide developers with an ability to specify the relative importance of particular deadline-constrained threads. For example, though there may be two threads with the same deadline, meeting the deadline for the first of these threads might be considered more important than meeting the deadline for the second.

Negotiating Component

A negotiating component (NC), is a “place holder” name for a type of component that implements a set of special interfaces (maybe in the technical Java sense of the word) that let it communicate with the Java platform about its resource requirements. It is not yet clear to the Requirements Group how this should done, but it will probably involve budget requests from the component to the Java Platform, offers and error messages coming back down, and other interactions.

If an NC exceeds a resource budget it should be told (perhaps by an event or an exception), and the component that is responsible for the erring component (perhaps the component’s installer or invoker) should be notified as well. It should at least be able to request notification. The most extreme option would have all notifications directed to the responsible component. The erring component could be notified by the responsible component (or not).

Terms

An **Ahead of time (AOT) compiler** transforms a body of code (e.g., source code, bytecode) into native code for future execution within a runtime environment.

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 tried 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.

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 byte-code 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: Process that automatically identifies dynamically allocated objects that are no longer reachable (garbage) and reclaims the space occupied by the objects. Garbage collection consists of both garbage detection and garbage reclamation.

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

A **Just-in-time (JIT) compiler** is used by an interpreter to, on-the-fly, transform a body of code (e.g., bytecode) into native code for execution within an interpreted runtime.

Negotiating component (NC): a “place holder” name for a type of component that implements a set of special interfaces (maybe in the technical Java sense of the word) that let it 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.

The **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.

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

The Sandbox comprises a number of cooperating system components, ranging from security managers that execute as part of the application, to security measures designed into the JVM and the language itself. The sandbox ensures that an untrusted, and possibly malicious, application can not gain access to system resources. [Javasoft]

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

The Virtual machine is an abstract specification for a computing device that can be implemented in different ways, in software or hardware. … **The Java Virtual Machine** consists of a bytecode instruction set, a set of registers, a stack, a garbage-collected heap, and an area for storing methods. [Javasoft]

Sources

[Javasoft] Sun Microsystems provides an online glossary of Java related terms. See http://www. Javasoft.com/docs/glossary.print.htm).

[Jensen] Doug Jensen, Mitre Corp., 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(tm) Traits

The Requirements WG considers these Java Traits to provide a basis for the real-time requirements. (consensus – agree)

- 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 by keeping software components (including the VM itself) protected from one another.
- Java supports dynamic loading of new classes.
- Java is highly dynamic; supporting lots of object/thread creation at run time.
- 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 language and platforms support application portability.
- The Java technologies support distributed applications.
- Java provides well-defined execution semantics.

Section 3 Guiding Principles

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

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

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

Section 4 Goals and Derived Requirements

EDITOR'S NOTES:

1. Many of these requirements are written in terms of providing functionality, capabilities to ***application programmers****. The WG should consider the underlying (and sometime hidden) 'requirements' that would be necessary to provide these capabilities. Additionally, the WG should consider using these as 'Criteria to Evaluate'.*

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/predictibility trade-offs). (consensus – agree)

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

Derived Requirements:

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. (consensus – agree)

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

---

[1] RTJ is used to identify the sum of the functionality and services that would be provided through real-time extensions to the Java™ Platform.

DR 2.3 Standard subsets of RTJ and RTJVM specification should be created as necessary to support improved efficiency and/or reliability for particular specialized domains (consensus –agree) *ED NOTE: The WG needs to define these domains. Possible candidates: ard real-time safety critical software, distributed systems, embedded systems. The WG needs to decide how to slice the pie.*

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. (consensus – agree)

Derived Requirements:

DR 3.1 Minimal human intervention should be required when the software is "ported" to new hardware platforms or combined with new software components. (consensus – agree)

DR 3.2 RTJ should abstract operating system and hardware dependencies. (consensus – agree)

DR 3.3 RTJ must support standard Java semantics. (consensus – agree)

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

GOAL 4: RTJ should support workloads comprised of the combination of real-time tasks and non-real-time tasks. (consensus –agree)

Derived Requirements:

DR 4.1 Traditional Java software must run as non-real-time tasks. (consensus –agree)

DR 4.2 Facilities must be provided to allow sharing of information between non-real-time and real-time tasks. (consensus –agree)

DR 4.3 The sharing protocols must provide the real-time programmer with the ability to avoid unpredictable blocking delays. (consensus –agree)

DR 4.4 The relationships between RTJ threads with the other three possible types of threads (non-real-time Java, non-Java real-time, and non-Java non-real-time) need to be formally defined. (consensus – agree) *ED NOTE: Thread relationship paragraphs and model will be inserted here.*

GOAL 5 : RTJ should allow real-time application developers to separate concerns between negotiating components. (consensus – agree)

GOAL 6: RTJ should allow real-time application developers to automate resource requirements analysis either at runtime or offline. (consensus – agree)

GOAL 7: RTJ should allow real-time application developers to write real-time constraints into their software. (consensus with serious reservations regarding the ramifications of these clauses.

Derived Requirements

DR 7.1 RTJ should provide application developers with the option of using conservative or aggressive resource allocation. (consensus – open)

DR 7.2 The same RTJVM should support combined workloads in which some activities budget aggressively and other conservatively. (consensus – open)

DR 7.3 RTJ infrastructure should allow negotiating components to take responsibility for assessing and managing risks associated with resource budgeting and contention. (consensus-agree)

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. (consensus – agree)

GOAL 8: Real-time Java should allow resource reservations and should enforce resource budgets. The following resources should be budgeted: CPU time, memory and memory allocation rate. (consensus – agree) *ED NOTE: definition for memory allocation rate*

GOAL 9: RTJ should support the use of components as "black boxes"; including on the same thread. (consensus - open)

DR 9. 1: RTJ should support dynamic loading and integration of negotiating components. (consensus - agree)

DR9.2: RTJ should support a mechanism for negotiating components whereby the behavior of critical sections of code is locally analyzable. (consensus- open)

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 definition of "real-time garbage collection" for further discussion. (consensus –agree) *ED NOTE: Specify what constitutes GC implementation 'information'.*

GOAL 11: RTJ should support straightforward and reliable integration of independently developed software components. This includes changing hardware. (consensus –agree)

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

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

Section 5 Resource Management (RM) Requirements

Introductory Discussion

Example resources include:

- CPU resources raising issues of scheduling, preemption, fairness, time slicing;
- Exclusive access to shared data using synchronized block/method

- Access to restricted (user-defined) resources using wait/notify
- Network routines using RMI

The issues that these requirements address: *ED NOTE: This section must be detailed.*

- Deadlock avoidance
- Priority inversion
- Budgeting in Advance
- Static availability
- Global vs. local knowledge
- Deadline vs. priority
- Sytem knowledge vs. application knowledge
- Finalizers: how to ensure timeliness and correctness

Utility of Resource Management:

The following capabilities should be supplied or be implementable using the functionality delineated in the specification:

- dead-line based scheduler on top of (or perhaps within) RTJVM. (consensus –agree)
- time-sliced scheduler or periodic scheduler on top (or perhaps within), (optionally make time-slicing a required option.) (consensus –agree)
- resource accounting (especially CPU accounting.) (consensus –agree)
- The ability to determine maximum defferal of event due to primitives that block event. (In what units?) (consensus –open)

ED NOTE: Do the requirements below support these capabilities?

Derived Requirements

RM DR1: The RTJ API must be well-defined with guarantees on all language features. (consensus – agree)

RM DR1 is based on 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. (consensus – agree)

RM DR 2: Ability to enforce (with notification, event handling and accounting) space/time limits, in a scoped manner, from the outside; on "standard" Java features too. (consensus – agree*)*

RM DR2 is based on GOAL 9: RTJ should support the use of components as "black boxes"; including on the same thread. (consensus –open)

RM DR3: In a real-time context, existing Java features shall "work right" including 'synchronized' (bounded priority inversion) and 'wait/notify' (priority queuing). (consensus – agree)
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.

RM DR3 is based on GOAL 9: RTJ should support the use of components as "black boxes"; including on the same thread.

RM DR 4: RTJ must:

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

RM DR4 is based on 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. (consensus – agree).

Open Issues

1. What to do with 'schedulability analysis'? Requires more discussion. (Roadmap requirement)

Section 6 Requirements for Memory Management (MM)

Introductory Discussion

The following capabilities should be implementable using the functionality delineated in the specification:

- The ability to analyze code to determine memory consumption. (e.g., How much garbage is created and how much ‘non-garbage’ is created.) (consensus – open) *ED NOTE: Do the requirements below support this capability?*
- The ability to specify rate ‘hints’ for GC that would be paced according to the rate that memory is allocated. (consensus – open) *ED NOTE: (1) Do we all know what is meant by ‘hints’? (2) Do the requirement below support this capability?*
- The ability to partition/budget memory between components (e.g., Component A get memory regardless of what component B wants.) (consensus – open) *ED NOTE: (1) Is plain old ‘components’ appropriate here? (2) Do the requirements below support this capability?*

Derived Requirements

MM DR1: Language and libraries are clearly understood in terms of memory usage. (consensus – agree) *ED NOTE: This requirement must be further refined to be useful.*

MM DR1 is based on Goal 8: Real-time Java should allow resource reservations and should enforce resource budgets. The following resources should be budgeted: CPU time, memory and memory allocation rate. (consensus – agree)

MMDR2: RTJ defines ‘garbage’. See definition of ‘garbage’ for futher discussion. (consensus – agree)

MMDR2 is based on Goal 10: RTJ must provide real-time garbage collection when garbage collection is necessary. GC implementation 'information' must be visible to the RTJ application.

MMDR3: RTJ should provide ‘hint handling’ information regarding the GC, (e.g., accurate vs. conservative ? Does it defragment?). (consensus – open) *ED NOTE: Paragraphs defiining the specific information to be inserted here.*

MMDR3 is based on Goal 10: RTJ must provide real-time garbage collection when garbage collection is necessary. GC implementation 'information' must be visible to the RTJ application.

MMDR4: RTJ must not restrict nor specify the garbage collection technique; rather it should be capable of supporting all appropriate techniques for RT GC. (consensus – agree)

MMDR4 is based on Goal 10: RTJ must provide real-time garbage collection when garbage collection is necessary. GC implementation 'information' must be visible to the RTJ application.

MMDR5: The GC must make forward progress at some rate. The rate must be 'queryable' and configurable. (consensus – agree) *ED NOTE: Paragraph further defining 'forward progess' (i.e., the 'incremental discussion' to be inserted here.*

MMDR5 is based on Goal 10: RTJ must provide real-time garbage collection when garbage collection is necessary. GC implementation 'information' must be visible to the RTJ application.

MMDR6: RTJ shall provide support for a guaranteed allocation rate. (consensus – agree)

MMDR6 is based on 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.

MMDR7: RTJ must/should not require bounds on when an object is finalized or reclaimed. (consensus – agree)

MMDR7 is based on GOAL 8: Real-time Java should allow resource reservations and should enforce resource budgets. The following resources should be budgeted: CPU time, memory and memory allocation rate. (consensus – agree)

MMDR8: Within RTJ the mutator overhead must be bounded if GC is in process. (consensus – agree)

MMDR8 is based on Goal 10: RTJ must provide real-time garbage collection when garbage collection is necessary. GC implementation 'information' must be visible to the RTJ application.

MMDR9: RTJ should provide for specifying memory.

MMDR9 is based on 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.

Open Issues

1. Finalizers (timeliness and correctness)

2. Stack-based allocation. Automatic or explicit?
3. Reference count GC.
4. Definition and use of the term ‘incremental’ for GC.

Section 7 Asynchrounous Event Handling

ED NOTE: Description of Model for Asynchronous Event Handling to be inserted here

Rationale for Model:

1. To stop threads (timeouts)
2. For refinement computations
3. Change in execution environment
4. Polling (a different model) requires global knowledge
5. Shutdown (for fail safes)
6. Overload situations
7. Would help support Ada
8. Converting hardware interrupts to synchronous events

When would event handling be deferred?

- Not in synchronized code; however its lock should be cleared.
- In ‘finally’ clause.
- Not in ‘catch’ clause.
- In user-defined locations (e.g., atomic {…}).
- Not in ‘finalize’ method.
- User defined.