
# Doing Real-Time With Java

**Kelvin Nilsen, Chair of the Real-Time Java Working Group**

# Why Real-Time Java is relevant?


- Java is replacing C, C++, Pascal as the preferred undergraduate instructional language.
- Real-Time Java threatens to displace Ada as the preferred Defense Department language.
- Real-Time Java offers the potential of improving the state of the art for integration of real-time software components.
- Different approaches are appropriate for different needs (small devices vs. large systems).
- Near-term commercial requirements are for “que será, será” soft real time.

# History


- 1988: dissertation on high-level real-time programming language features, including real-time garbage collection
- 1988-1996: additional research on real-time garbage collection, culminating in attempts to commercialize for C++
- May 1995: Sun released Java
- Dec 95: “Issues in the Design and Implementation of Real-Time Java”, followed by a draft specification to address the issues

# History


- March 96: Nilsen founded NewMonics, to focus on commercialization of real-time Java technologies
- Summer 96: RTOS companies (Microware, Wind River) licensed Java from Sun
- Fall 97: NIST workshop on Java conformity assessment. Nilsen represented “Real Time”
- June - Dec 98: NIST series of Real-Time Java workshops (involving 37 companies)

# History


- Nov 98: RTJWG formed to create specification
- Feb 99: Sun formed RT Expert Group to work on RTSJ
- Oct 00: J Consortium (sponsor of RTJWG) approved as ISO PAS submitter

# NIST Requirements


- 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).”

# NIST Goals


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

# Derived Requirements: 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 subset of RTJ and RTJVM specifications should be created as necessary to support improved efficiency and/or reliability for particular specialized domains.”

# NIST Goals


- 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: 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.”

# NIST Goals


- 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.”

# NIST Goals


- Goal 6: “RTJ should allow real-time application developers to automate resource requirements analysis either at run-time or off-line.”
- Goal 7: “RTJ should allow real-time application developers to write real-time constraints into their software.”

# Derived Requirements: Goal 7


- DR 7.1: “RTJ should provide application developers with the option of using conservative or aggressive resource allocation.” [no consensus]
- DR 7.2: “The same RTJVM should support combined workloads in which some activities budget aggressively and others conservatively.”
- DR 7.3: “RTJ infrastructure should allow negotiating components to take responsibility for assessing and managing risks associated with resource budgeting and contention.”

# Derived Requirements: Goal 7


- 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.”

# NIST Goals


- 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: 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.

# Derived Requirements: Goal 8


- DR 8.1 (continued):
  - 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.”

# Derived Requirements: Goal 8


- 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.”

# NIST Goals


- Goal 9: “RTJ should support the use of components as ‘black boxes’; including such use on the same thread.” [no consensus]

# Derived Requirements: 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.”

# Derived Requirements: Goal 9


- 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).”

# NIST Goals


- Goal 10: “RTJ must provide real-time garbage collection when garbage collection is necessary. GC implementation information must be visible to the RTJ application.”
  - RTGC has bounded system pause time, guaranteed rate of memory reclamation, and bounded allocation time

# Derived Requirements: 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? Defragmenting?)” [no consensus]
- DR 1.03: “RTJ must not restrict nor specify the garbage collection technique; rather, it should be capable of supporting all appropriate techniques for real-time GC.”

# Derived Requirements: Goal 10


- 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.”

# NIST Goals


- 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 other languages, such as Ada.
- Goal 13: “RTJ should be implementable on operating systems that support real-time behavior.”

# Alternative RTJ Approaches


- JNI Programming (as defined by Sun Java specification)
- Real-Time Specification for Java (by Sun's Java Community Process)
- Real-Time Core (by J Consortium)
- High-Level Profile of the Real-Time Core (by J Consortium)
- Real-Time Virtual Machine (by independent clean-room vendors)

# Currently Available Approaches


- JNI Programming (as defined by Sun Java specification)
- Real-Time Specification for Java (by Sun's Java Community Process)
- Real-Time Core (by J Consortium)
- High-Level Profile of the Real-Time Core (by J Consortium)
- Real-Time Virtual Machine (by independent clean-room vendors)

# Which is “Easiest” to Use?

CONSORTIUM

- Point and Shoot
  - Auto-everything
- SLR
  - Manual or auto focus, manual or auto exposure
  - Interchangeable lenses
- View Camera
  - Manual focus and exposure
  - Interchangeable lenses
  - Leaf shutters
  - Tilts, shifts, swings
  - Monstrous negatives



# Use the Right Tool for the Job


- Don’t expect:
  - Ansel Adams to find it “easy” to work with an SLR or point-and-shoot camera
  - A photo-journalist to work with a point-and-shoot or view camera
  - A snapshot enthusiast to “enjoy” an SLR
- Why do we expect software engineering professionals to be less competent than today’s professional photographers?
  - Do we really think a single tool (i.e. “language”) is best for every application?

# Photography Infrastructure


- Note that cameras share:
  - Same film and print paper technologies
  - Same processing chemistry
  - Same enlarge and reprint equipment
  - Same exposure control (shutter speed and aperture)
  - Same principles of operation: focus, depth of field, lens design, light metering, studio lighting

# Criteria for Comparisons


- Efficiency
- Predictability
- Latency
- Reliability
- Standardization
- Ease of Development
- Expressive Power
- Portability
- Scalability and Ease of Integration

# Efficiency


| | JNI | RTSJ | Core | Core Profile | RTVM |
|---|---|---|---|---|---|
| Efficiency | 2 | | | | |

JNI is generally much less efficient than solving the whole problem in Java, and is 3-4 times less efficient than C. – Jack Andrews of Space-Time Research

# Efficiency


| | JNI | RTSJ | Core | Core Profile | RTVM |
|---|---|---|---|---|---|
| Efficiency | 2 | 2 | | | |

RTSJ has a smaller, more specialized audience than “desktop Java”, and requires extra run-time checks each time an object field is read or written.

# Efficiency


| | JNI | RTSJ | Core | Core Profile | RTVM |
|---|---|---|---|---|---|
| Efficiency | 2 | 2 | | 3 | 3 |

RTVM and Core Profile are essentially desktop Java with real-time garbage collection. Making garbage collection real-time incremental imposes run-time penalties compared with desktop Java.

# Efficiency


| | JNI | RTSJ | Core | Core Profile | RTVM |
|---|---|---|---|---|---|
| Efficiency | 2 | 2 | 4 | 3 | 3 |

Core features (e.g. no garbage collection, no dynamic initialization/resolution, stack allocation) were designed to achieve efficiency comparable to C++, running faster and smaller than desktop Java.

# Predictability


| | JNI | RTSJ | Core | Core Profile | RTVM |
|---|---|---|---|---|---|
| Predictability | 3 | 3 | | | |

With both JNI and RTSJ, predictability is entirely a quality of implementation issue. Application code is predictable to the extent that the VM and underlying operating system are predictable.

# Predictability


| | JNI | RTSJ | Core | Core Profile | RTVM |
|---|---|---|---|---|---|
| Predictability | 3 | 3 | 4 | 4 | 4 |

With Core, Core Profile, and RTVM, predictability is part of the specified behavior. RTVM specifies fixed priorities, no priority aging, and priority inheritance. Latencies are implementation-defined (depends on platform). Conformance assessment requires certain predictability guarantees.

# Latency


| | JNI | RTSJ | Core | Core Profile | RTVM |
|---|---|---|---|---|---|
| Latency | 4 | 4 | 4 | | |

JNI, RTSJ, and Core are all designed to provide the “best possible” latency available on a given platform (hardware and operating system combination). Expect context switch latencies of less than 10 microseconds on typical state-of-the-art systems.

# Latency


| | JNI | RTSJ | Core | Core Profile | RTVM |
|---|---|---|---|---|---|
| Latency | 4 | 4 | 4 | 3 | 3 |

Core Profile and RTVM have slightly worse latencies, because time is required to preempt real-time garbage collection activities, and because task scheduling decisions are slightly more complicated than with JNI, RTSJ, and Core. Expect latencies of 100-200 microseconds on state-of-the-art platforms.

# Reliability


| | JNI | RTSJ | Core | Core Profile | RTVM |
|---|---|---|---|---|---|
| Reliability | 1 | | | | |

JNI has all the reliability weaknesses of C, which are amplified by the complexity of the JNI object sharing protocols. It is far too easy for programmers to make mistakes, and the consequences of their mistakes are far reaching.

# Reliability


| | JNI | RTSJ | Core | Core Profile | RTVM |
|---|---|---|---|---|---|
| Reliability | 1 | 2 | 2 | | |

Stylized Java is much less error prone than C and/or assembly language, because RTSJ and Core build on the strong type system of Java. Also, these technologies provide better memory management protection than C through run-time checks (RTSJ) or static checks (Core). However, the level of detail required of programmers and the impact of mistakes make these less reliable than desktop Java.

# Reliability


| | JNI | RTSJ | Core | Core Profile | RTVM |
|---|---|---|---|---|---|
| Reliability | 1 | 2 | 2 | | 3 |

RTVM offers reliability comparable to desktop Java implementations in that it provides automatic garbage collection, secure dynamic loading (enforcement of type system), and security management capabilities. RTVM programmers are prevented from crashing the operating system, for example.

# Reliability


| | JNI | RTSJ | Core | Core Profile | RTVM |
|---|---|---|---|---|---|
| Reliability | 1 | 2 | 2 | 4 | 3 |

Core Profile offers additional reliability benefits, such as partitioning of memory and CPU time between particular components.

# Standardization


| | JNI | RTSJ | Core | Core Profile | RTVM |
|---|---|---|---|---|---|
| Standardization | 3 | 3 | | | 3 |

JNI, RTSJ, and RTVM all adhere to “de facto standards” defined by Sun Microsystems and “widely adopted” across many industries. Specifications are published, but ambiguous and/or incomplete. Compatibility testing is provided by Sun Microsystems under special license terms and by independent third parties (e.g. Plum Hall, Perennial).

# Standardization


| | JNI | RTSJ | Core | Core Profile | RTVM |
|---|---|---|---|---|---|
| Standardization | 3 | 3 | 4 | 4 | 3 |

Core and Core Profile are defined by the J Consortium, and (presumably) approved by ISO as international standards. The specifications are more thorough, and ambiguities are resolved through open, consensus-based procedures.

# Ease of development


| | JNI | RTSJ | Core | Core Profile | RTVM |
|---|---|---|---|---|---|
| Ease of development | 1 | | | | |

JNI is particularly difficult to develop with. Programmers face all of the challenges of traditional C or C++, combined with the complexity of using the very low-level and error-prone JNI protocols.

# Ease of development


| | JNI | RTSJ | Core | Core Profile | RTVM |
|---|---|---|---|---|---|
| Ease of development | 1 | 2 | 2 | | |

RTSJ and Core are easier than JNI, because all development is done in the same language (Java), which has a strong type system. But the protocols required of these programmers are fairly low level and detail oriented, making development more difficult than desktop Java development.

# Ease of development


| | JNI | RTSJ | Core | Core Profile | RTVM |
|---|---|---|---|---|---|
| Ease of development | 1 | 2 | 2 | | 3 |

RTVM offers the same ease of development as traditional desktop Java, including real-time garbage collection. Developers report approximately two-fold productivity improvement over C++.

# Ease of development


| | JNI | RTSJ | Core | Core Profile | RTVM |
|---|---|---|---|---|---|
| Ease of development | 1 | 2 | 2 | 4 | 3 |

With alternative approaches, every developer of a real-time component must “worry” about what every other real-time component is doing with memory, CPU time, and resource locking. Core Profile builds upon the strengths of RTVM, adding the capability to encapsulate resource requirements within software components.

# Expressive Power


| | JNI | RTSJ | Core | Core Profile | RTVM |
|---|---|---|---|---|---|
| Expressive power | 2 | | | | |

JNI provides no way to describe timeouts, deadlines, periods of execution, partitioning of memory or CPU time, etc. The semantics of task priorities and synchronization are not specified.

# Expressive Power


| | JNI | RTSJ | Core | Core Profile | RTVM |
|---|---|---|---|---|---|
| Expressive power | 2 | | | | 3 |

RTVM allows programmers to speak of “deterministic” priorities and to define synchronized blocks which implement priority inheritance.

# Expressive Power


| | JNI | RTSJ | Core | Core Profile | RTVM |
|---|---|---|---|---|---|
| Expressive power | 2 | 4 | 4 | | 3 |

RTSJ and Core add support for asynchronous transfer of control, timeouts, priority ceiling protocol, and increased numbers of thread priorities.

# Expressive Power


| | JNI | RTSJ | Core | Core Profile | RTVM |
|---|---|---|---|---|---|
| Expressive power | 2 | 4 | 4 | 5 | 3 |

Core Profile adds deadline-driven and benefit-based scheduling, memory and CPU time partitioning, and workload balancing.

# Portability


| | JNI | RTSJ | Core | Core Profile | RTVM |
|---|---|---|---|---|---|
| Portability | 2 | | | | |

JNI is no more portable than the C language and RTOS services upon which it depends.

# Portability


| | JNI | RTSJ | Core | Core Profile | RTVM |
|---|---|---|---|---|---|
| Portability | 2 | 3 | | | |

RTSJ benefits from the portability benefits of the Java language, but exhibits RTOS and VM dependencies.

# Portability


| | JNI | RTSJ | Core | Core Profile | RTVM |
|---|---|---|---|---|---|
| Portability | 2 | 3 | 4 | 4 | 4 |

Core, Core Profile, and RTVM provide consistent behavior across CPU and operating system platforms.

# Scalability and Ease of Integration


| | JNI | RTSJ | Core | Core Profile | RTVM |
|---|---|---|---|---|---|
| Scalability and ease of integration | 1 | | | | |

Integrating JNI components is difficult because programmers must avoid C global variable name conflicts, manually isolate variables and services to restricted scopes, prevent resource sharing conflicts, and deal with operating system incompatibilities.

# Scalability and Ease of Integration


| | JNI | RTSJ | Core | Core Profile | RTVM |
|---|---|---|---|---|---|
| Scalability and ease of integration | 1 | 2 | | | |

RTSJ prevents global naming conflicts, and uses Java's strong type system to limit the visibility of private variables and services.

# Scalability and Ease of Integration


| | JNI | RTSJ | Core | Core Profile | RTVM |
|---|---|---|---|---|---|
| Scalability and ease of integration | 1 | 2 | | | 3 |

RTVM builds upon the benefits of RTSJ by eliminating operating system dependencies.

# Scalability and Ease of Integration


| | JNI | RTSJ | Core | Core Profile | RTVM |
|---|---|---|---|---|---|
| Scalability and ease of integration | 1 | 2 | 4 | | 3 |

Core enables watchdog tasks to monitor tasks and abort those that are misbehaving, in addition to the benefits of RTVM.

# Scalability and Ease of Integration


| | JNI | RTSJ | Core | Core Profile | RTVM |
|---|---|---|---|---|---|
| Scalability and ease of integration | 1 | 2 | 4 | 5 | 3 |

Core Profile adds services to support partitioning of memory and CPU time between components, and to limit the time that tasks can place exclusive locks on shared resources.

# Tabular Summary


| | JNI | RTSJ | Core | Core Profile | RTVM |
|---|---|---|---|---|---|
| Efficiency | 2 | 2 | 4 | 3 | 3 |
| Predictability | 3 | 3 | 4 | 4 | 4 |
| Latency | 4 | 4 | 4 | 3 | 3 |
| Reliability | 1 | 2 | 2 | 4 | 3 |
| Standardization | 3 | 3 | 4 | 4 | 3 |
| Ease of development | 1 | 2 | 2 | 4 | 3 |
| Expressive Power | 2 | 4 | 4 | 5 | 3 |
| Portability | 2 | 3 | 4 | 4 | 4 |
| Scalability | 1 | 2 | 4 | 5 | 3 |

# Summary


- Different technologies support different needs
  - JNI: for quick & dirty, small, “portable” real-time components
  - RTSJ: similar, with better performance, but only works with “non-standard” Java virtual machines
  - Core: real-time infrastructure code, low level, high performance
  - Core Profile: large, complex real-time systems, dynamic behavior, high-level programming
  - RTVM: easy integration of reusable non-real-time components, medium-complexity real-time systems

# How to Help?


- If industry and government truly desire open standards and conformance, you must participate:
  - Review draft specifications and provide feedback
  - Help fund development of prototype implementations and trial case studies
  - Guide the development of certification and branding programs that suit your objectives