



Thoughts Regarding Sun's Real-Time Specification for Java

Kelvin Nilsen

**The Real-time and Embedded Systems Forum
The Open Group
January 24, 2001**

NewMonics



Key Ideas

- The Real-Time Specification for Java (RTSJ) represents the accumulation of a lot of hard work by many smart people.
- You can definitely write real-time software using the RTSJ.
- Does RTSJ represent an improvement over the state of the art?
 - This depends on your perspective

Differing perspectives

- Low-level “system” software vs. high-level application software
- Programming in the large vs. programming in the small
- Experimental research
- Safety critical systems

Low-Level System Software

- Operating System Examples: device drivers, network protocol stacks, schedulers and dispatchers, synchronization
- Application Examples: radar, sonar, weapons, battery-powered devices, flight surfaces
- Requirements: high performance, small footprint, well-understood semantics, reliability, hard and soft real-time
- Desirables: portability, clean integration

High-Level Application Software

- Examples: Network Infrastructure, C2, C4SI, Air Traffic Control, AWACS, Ballistic Missile Defense, Training simulators, Video Games
- Requirements: Well-understood semantics, Ease of development, Ease of integration, Portability, Reliability, Soft real time
- Desirables: Efficient use of memory and CPU

Programming in the Small

- Examples: play piano with robot arm, dueling or dancing robots, data acquisition in research lab, ladder logic industrial automation applications
- Comments: not essential to have well-understood semantics, portability, or integration – one programmer does development for a single platform, and experiments as he works
- But this developer wants a small and simple development system

Experimental Research

- Examples: Studying real-time scheduling, synchronization, distributed programming, real-time garbage collection
- Requirements: maximal generality and flexibility
 - This conflicts with stability desired for large-scale or safety critical development

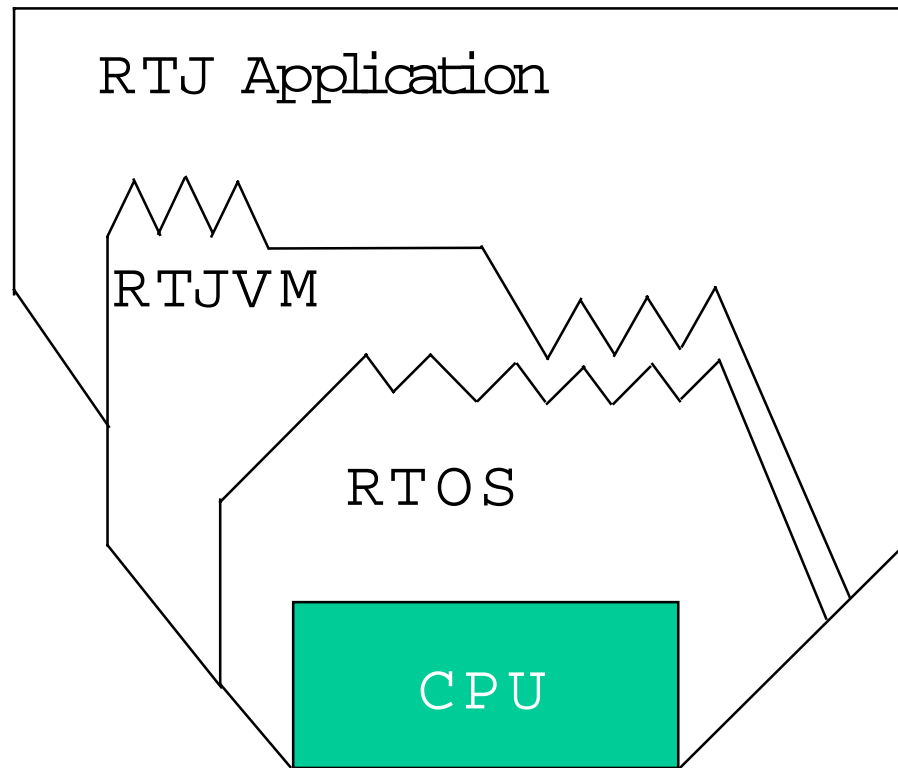
Safety Critical Systems

- Examples: Commercial aerospace, nuclear power plants, automation of rail transportation systems
- Requirements: Small and simple software architectures, extreme conservatism, certification by auditing agency

Fundamental RTSJ Limitations

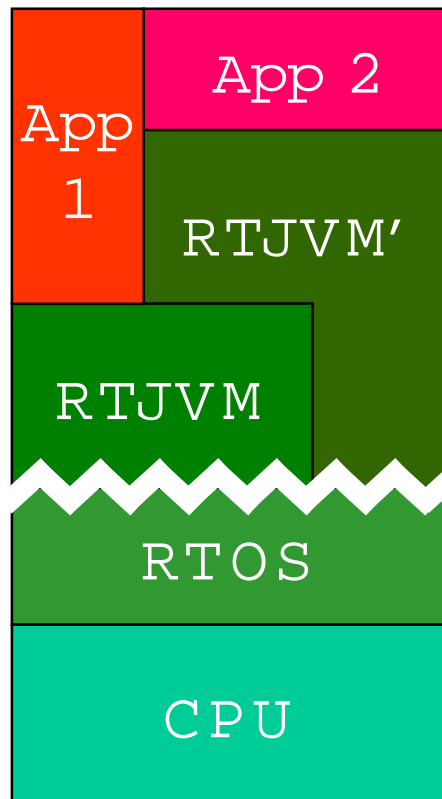
- Paradigm mismatch
- Complexity
 - Hard to understand, validate
- Ambiguity
 - Hinders portability, scalability, reuse
 - Increases costs of development and maintenance
 - Difficult to integrate independently developed components

RTSJ Programming Model



RTSJ specifies that the APIs remain the same. Only the semantics handle

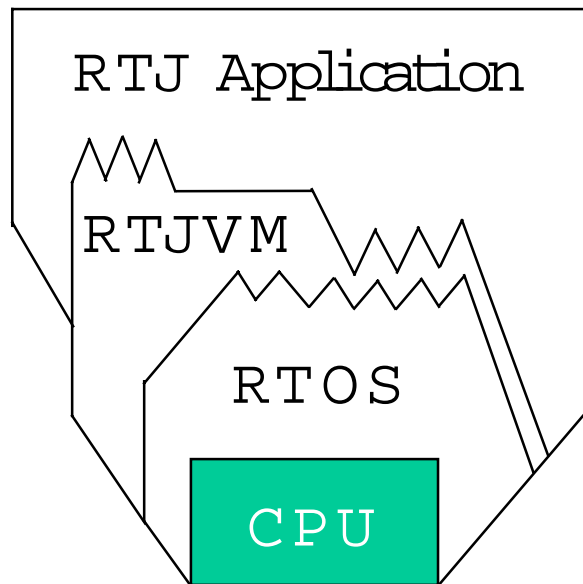
My Preferred Extensible Architecture



“Therefore whosoever heareth these sayings of mine, and doeth them, I will liken him unto a wise man, which built his house upon a rock:

And the rain descended, and the floods came, and the winds blew, and beat upon that house; and it fell not: for it was founded upon a rock.”

Personal Opinion, Continued...



“And every one that heareth these sayings of mine, and doeth them not, shall be likened unto a foolish man, which built his house upon the sand:

And the rain descended, and the floods came, and the winds blew, and beat upon that house; and it fell: and great was the fall of it.” – Matthew 7: 24-27

Talk Roadmap

- Ambiguity in the RTSJ
- Conflicts between requirements and design of the RTSJ
 - Requirements depend on perspective
- Summary

Sources of Ambiguity

- Scheduler Behavior
- Synchronization Behavior
- Memory Management
- Library Differences
- Other Surprises

Ambiguity Index

- Some sources of ambiguity may be unintentional oversights in the specification
 - Though it's hard to tell, since the intents and philosophies of the Real-Time Experts Group are not always clear
- Other sources of ambiguity appear to be by design

Scheduling Ambiguity (Design)

- All systems implement the “base scheduler”:
28 priorities, fixed priority, preemptive
 - “Implementations providing additional scheduling policies or execution eligibility assignment policies which require an application visible field to contain execution eligibility then SchedulingParameters must be subclassed...” – p. 46
 - How does the presence of tasks running under a secondary scheduler impact tasks running under the primary scheduler?

Scheduling Ambiguity (Design)

- “Threads that are preempted in favor of a higher priority thread may be placed in the appropriate queue **at any position** as determined by a particular implementation. **The implementation is required to provide documentation** stating exactly the algorithm used for such placement.” – p. 47

Scheduling Ambiguity

(Design and perhaps oversight)

- ***“Any method of constructor that accepts a RationalTime of (x,y) must guarantee that its activity occurs exactly x times in every y milliseconds, even if the intervals between occurrences of the activity have to be adjusted slightly. The RTSJ does not impose any required distribution on the lengths of the intervals but strongly suggests that implementations attempt to make them of approximately equal lengths.” – p. 148***
 - *What if y is not a multiple of the timer tick?*
 - *Satisfying this requirement implies busy wait loops in the scheduler and/or dispatcher! Maybe they didn't really mean this?*

Scheduling Ambiguity

“In some real-time systems, there may be a large number of potential events and event handlers (numbering in the thousands or perhaps even the tens of thousands), although at any given time only a small number will be used. Thus it would not be appropriate to dedicate a thread to each event handler. The RTSJ addresses this issue by allowing the programmer to specify an event handler either as not bound to a specific thread (`AsyncEventHandler`) or alternatively as bound to a thread (`BoundAsyncEventHandler`).”

Scheduling Ambiguity

(Is this a design issue, or oversight?)

- For unbound handlers:
 - How much memory is allocated for event handler?
Where? When?
 - Does this delay startup of event handler? By how much?
 - Is a fixed-size thread pool a conforming implementation?
 - If so, can blocking event handlers deadlock?
 - If not, might handling of async event exceed number of “system” threads?

Scheduling Ambiguity

“Caution: `<string>`Parameter objects are explicitly unsafe in multithreaded situations when they are being changed. No synchronization is done. It is assumed that users of this class who are mutating instances will be doing their own synchronization at a higher level.” – p. 19

“When a reference to a Parameters object is given as a parameter to a constructor, the Parameters object becomes bound to the object being created. Changes to the values of the Parameters object affect the constructed object.” – same page

Scheduling Ambiguity

(1. documentation oversight, 2. design)

- Thread scheduling, dispatching, and execution are inherently asynchronous with respect to each other. How do they synchronize?
- If we change scheduling parameters, when do the changes take effect? Does it depend on where we are in the current execution period? What happens during the transition?

Scheduling Ambiguity (design?)

ReleaseParameters(RelativeTime cost,
RelativeTime deadline, AsyncEventHandler
overrunHandler, AsyncEventHandler
missHandler)

- “Although minimum implementations do not consider deadlines in feasibility calculations, they must recognize variable deadlines and invoke the miss handler as appropriate.” – p. 55
- Exactly when is the miss handler triggered?

Ambiguity in Synchronization

(Not sure: design or oversight?)

- “Determinism: Conforming implementations shall provide a fixed upper bound on the time required to enter a synchronized block for an unlocked monitor.” – p. 11
 - What about the time required to exit a monitor? No statement in the specification.

Synchronization Ambiguity (Design)

“setMonitorControl(policy) – control the default monitor behavior for object monitors used by synchronized statements and methods in the system. The type of the policy object determines the type of behavior. Conforming implementations must support priority ceiling emulation and priority inheritance for fixed priority preemptive threads.” – p. 137

- How quickly does synchronization policy change?***
- What happens during the transition?***
- Note how heavy this semantics is.***

Memory Subsystem Ambiguity (Design or oversight?)

```
public MemoryParameters(  
    long maxMemoryArea,  
    long maxImmortal, long allocationRate)  
    throws IllegalArgumentException
```

- What happens if my thread exceeds the allocation rate? Stalling? Run-time exception?

Memory Subsystem Ambiguity (oversight)

p. 34: “logic contained in [a NoHeapRealtimeThread’s] run() **is never allowed** to allocate or reference any object allocated in the heap nor is it even allowed to manipulate any reference to any object in the heap. For example, if a and b are objects in immortal memory, b.p is reference to an object on the heap, and a.p is type compatible with b.p, then a NoHeapRealtimeThread is not allowed to execute anything like the following:

```
a.p = b.p; b.p = null;”
```

- Who enforces this? How? Run-time exceptions? Does this imply a read barrier?
- If not enforced, system may crash.

Memory Subsystem Ambiguity (design or oversight?)

- “Each instance of the virtual machine will behave as if there is an area of memory into which all Class objects are placed and which is unexceptionally referenceable by NoHeapRealtimeThreads.” – p. 73
 - Does this mean NoHeapRealtimeThreads can always access static (class) variables?
 - Does this affect GC of classes?
 - May dynamic class loading consume immortal memory?
 - Are there restrictions on what I can read from class variables? (a la preceding slide)

Memory Subsystem Ambiguity (oversight?)

- “The dynamic scope cannot be reused until finalization is complete and the RTSJ requires that the finalizers execute to completion before the next use (calling `enter()` in a constructor) of the scoped memory” – p. 8.
 - How is this enforced? Blocking? Run-time exceptions?
 - Doesn't appear to be any way to check whether scope is ready to be reused.

Ambiguity in Library Behavior (design)

- How do I know which libraries allocate memory? How much memory?
- NoHeapRealtimeThread cannot read or write the heap. How do I know which standard libraries don't access the heap?
- NoHeapRealtimeThread cannot “manipulate” references to the heap. Which standard libraries manipulate references to the heap?
- Many traditional libraries attempt to build linked data structures comprised of existing and newly allocated objects, linked in ways that might violate dynamic scoping rule. Which libraries?

Ambiguity in Library Behavior (oversight?)

- “The RTSJ specifically requires that blocking methods in `java.io.*` must be prevented from blocking indefinitely when invoked from a method with AIE in its throws clause. The implementation, when either `AIE.fire()` or `RealtimeThread.interrupt()` is called when control is in a `java.io.*` **method invoked from an interruptible method, may either unblock the blocked call, raise an `IOException` on behalf of the call, or allow the call to complete normally if the implementation determines that the call would eventually unblock.**” – p. 177
 - If we unblock, what value do we return? What is resulting state?
 - If we throw an `IOException`, what is the resulting state of the I/O channel and buffer? (Open? Closed? Buffer contents?)

Other Sources of Ambiguity (design)

- “To be conformant to this specification, an implementation must provide documentation regarding the expected behavior of particular mechanisms. The mechanisms requiring such documentation, and the specific data to be provided, will be detailed in the class and method definitions.” - p. 19
 - Nice thought, but no such descriptions are provided.
 - Note that we need descriptions of all real-time methods, plus all traditional methods.

Other Sources of Ambiguity (design?)

- “ResourceLimitError exception is thrown if an attempt is made to exceed a system resource limit, such as the maximum number of locks” – p. 221
 - What other resource limits might be “hidden” within an implementation?
 - Total number of memory object handles?
 - Number of threads?
 - Number of network sockets?
 - Number of open files?

Other Sources of Ambiguity

- Licensing issues:
 - Using the RTSJ spec may be governed by onerous terms, conditions, or royalties
 - Sun license policies tend to evolve from one revision to the next

Other Sources of Ambiguity

- On 12-Nov-2001, Sun ABSTAINED FROM VOTING with the following comment: The effort behind JSR-001 is longstanding, predating the JCP. While there have been efforts to bring it in line with the spirit of the JCP as that evolved, there remain aspects that do not align with the JCP.
- For instance, the apparent fact that the RI license does not permit commercial use of the RI or its derivatives is contrary to the expectations of most JCP participants, for whom an RI is typically more than just a proof of concept. The TCK, while adequate in many areas, is known to have holes in its coverage, with well-understood consequences. The TCK is proposed to be licensed under the Community Public License. A CPL-licensed TCK leaves the door open for derivative TCKs that could confuse users and threaten WORA by raising the question whether an implementation is "more than compatible". To the extent portions of the RI are licensed under CPL, inadequate protection is given for stability within the Java platform name space.

The Paradigm Mismatch

- The question is whether the design of the RTJS, as published, addresses the needs of the market? Which market(s)?
- Consider the many sources of ambiguity in the API.
- Consider the intent to evolve the foundation to meet specific market niche requirements.
- No matter how careful you are, WOCRAC is probably not possible.

Examples of the Paradigm Mismatch

- Long-running systems with reliability and flexibility requirements probably need dynamic memory management that:
 - Supports shorter lifetimes than immortal
 - But does not necessarily fit LIFO behavior of scoped memory
 - Note: $FA < PDA < \text{Turing Machine}$

Examples of the Paradigm Mismatch

- The RT Experts Group avoided treatment of GC because they weren't sure how RT GC challenges would be addressed.
 - They left placeholders into which future implementers were to insert RT GC technologies.
 - But their placeholders are not adequate even to support solutions available today.
 - For example, they say: “NoHeapRealtimeThreads have an implicit execution eligibility that must be higher than that of the garbage collector.” This is fundamental to the RTSJ. However, given that **regular Java threads may never have an execution eligibility higher than the garbage collector, ...**” – p. 10
 - It seems clear that RTSJ does not intend for [Heap] RealtimeThreads to satisfy real-time constraints – consider lack of wait-free queues to communicate between RealtimeThreads and traditional Java threads.

Examples of the Paradigm Mismatch

- “PeriodicParameters.setIfFeasible(period, cost, deadline) returns true if, after considering the values of the parameters, the task set would still be feasible. In this case, the values of the parameters are changed. Returns false if, after considering the values of the parameters, the task set would not be feasible. In this case, the values of the parameters are not changed.” – p. 59
 - Very general, but not sufficient.
 - Assumes we can incrementally change system configuration, one thread’s parameters at a time
 - Many applications need a transaction-oriented interface (in which multiple threads change their scheduling parameters in a single atomic action)

Examples of the Paradigm Mismatch

- Note that Asynchronous Transfer of Control does not support “resumptive semantics”. Comments:
 - Often, we don’t know whether to abort or resume until after we have analyzed the circumstances surrounding the async exception. To avoid race conditions, we want to suspend ongoing efforts while we perform the analysis.
 - This is expensive, in that it requires an extra thread and/or increased/unpredictable latency if we use an unbound exception handler.
 - If all decision making is done by an asynchronous event handler, this adds considerable complexity to ensure synchronization and visibility of state changes.

Examples of Paradigm Mismatch

- In large real-time systems assembled of independently developed components, it is generally desirable to empower a centralized supervisor with control over all other activities. However, RTSJ allows individual threads to wrestle control away from centralized authority.
 - Many ways to mask asynchronous exceptions and termination, either accidentally or maliciously
 - Inner scopes may accidentally or maliciously catch and mask exceptions intended for outer scopes

Examples of Paradigm Mismatch

- Many aspects of RTSJ design obstruct high performance
 - No priority inversion avoidance between `NoHeapRealtimeThread` and other threads. Use wait-free queues and data copying instead.
 - Non-resumptive asynchronous exception handling.
 - Timeout set and cleared on every dispatch.
 - Dynamically adjustable scheduling and synchronization.
 - Non-standard “market niche” JVM.
 - Read and write barriers impact both code size and speed.

Summary

- The RTSJ is a large and complex work, representing the accumulation of many very bright people's efforts.
- However, in many regards, the RTSJ is lacking in elements of "good design".
- As a "standard", the RTSJ does not enable portability or interoperability.
- Additional work in identifying its target audience and tailoring the specification to better suit that audience would be beneficial.