Real-Time Java for Latency Critical Banking Applications

Bertrand Delsart
JavaRTS Technical Leader
Author of Sun's RTGC technology

Real-Time System

Agenda

- Background
- Benefits of a Real-Time Java Virtual Machine
- Benefits of Real-Time Garbage Collectors
- Q&A

Agenda

- Background
- Benefits of a Real-Time Java Virtual Machine
- Benefits of Real-Time Garbage Collectors
- Q&A

Banking Trend

- Avoiding latencies (unexpected delays) has always been important

Steve Rubinow (NYSE Group CTO):

“ If you've got some trades going through at 10 milliseconds and some at 1 millisecond, that's a problem. Our customers don't like variance”

Banking Trend

- Physical co-location is removing the external sources of jitter (variation of execution time)
- Application jitter is now much more visible

Dave Cummings (BATS CEO)

“Five years ago we were talking seconds, now we're into the milliseconds. Five years from now we'll probably be measuring latency in microseconds”

Alistair Brown (LIME Brokerage CEO)

“ Shortly, we'll be talking micro- versus milliseconds, and at that point speed will probably have less and less relevance”

Categories of Application Predictability

Non real-time	• No time-based deadlines > Standard programming logic > Example: UI, web services, algorithmic calculations
Soft real-time	• Deadlines may be missed occasionally under well-defined conditions > Mainstream real-time needs > Example: Automated trading system, 3G telco router
Hard real-time	• Deadlines cannot be missed > Typically highest priority on system > Example: Robotic motion control and management, Data feed processing

Traditional Design Choices to Improve Predictability

Use mainstream programming tools and standard platforms

OR

Use customized lower-level tools and proprietary platforms

- Everything is equally important; so try go faster to get everything done
- Guessing game: no guarantee that deadlines will be met – just postponed
- Low infrastructure utilization
- Lack of predictable solution

- Predictable solution, but...
- Expensive – dedicated tools, training, headcount; longer development cycles, complicated maintenance
- Challenging integration of real-time elements with the standard applications

Traditional Real-Time Design Choices: Examples

- Example 1: Global engineering conglomerate needs PLC control elements with hard real-time capabilities for industrial automation

PC-oriented SBC boards cost ~20% of custom boards but have no integrated real-time solution	OR	Custom boards from a specialized provider are very costly and require more for software/tools

- Example 2: Leading financial services institution needs to scale its market-facing system while meeting Service Level Agreements

Standard Java applications offer flexibility and scalability but GC pauses threaten SLAs	OR	Legacy system able to meet current SLAs but no longer viable with regard to maintainability and scalability

Real-Time Design Choices

Use mainstream program... stand...

Use customized lower-... and ... forms

How about an open standards-based platform that formally deals with real-time challenges?

> Everythin... importan... get every...
> Guessing ga... guarantee that deadlines will be met
> Low infrastructure utilization
> Lack of predictable solution

> ..., but... ...ted ...dcount, ...ent cycles, effort-intensive maintenance
> Challenging integration of real-time elements with the standard applications

Real-Time Specification for Java (RTSJ)

1998
Real-Time Specification for Java (JSR-001) proposal submitted
Joint Sun/IBM
Many others: Ajile, Apogee, Motorola, Nortel, QNX, Thales, TimeSys, WindRiver

2002
JSR-001 approved by the Java Community Process
TimeSys Reference Implementation

2005
RTSJ update proposal submitted (JSR-282)
- Apogee Aphelion
- Sun Java Real-Time System
- IBM's webSphere RealTime

2007
RTGC added to Sun Java RTS
Others companies implementing RTSJ (not yet certified)

2008
New Sun/IBM JSR

Agenda

- Background
- Benefits of a Real-Time Java Virtual Machine
- Benefits of Real-Time Garbage Collectors
- Q&A

Use case 1: Send Market Data at a Fixed Rate

Neither too late... nor too often !

- Mainstream Java solution:

```
while (true) do {
   compute_data();
   now = System.currentTimeMillis()
   Thread.sleep(next_period - now);
   send_data();
   next_period += period;
}
```

- Problem:
  - Not guaranteed to wake-up quickly after the sleep call !

RTSJ Benefit 1 : Priority Semantic

You must specify the relative importance of the tasks, including with respect to the other processes.

Mainstream "`setPriority(10);`"

is not sufficient !

- RTSJ offers in Java usual real-time scheduling semantics
  - > RealtimeThreads preempt non real-time ones
  - > Higher priority threads preempt lower priority ones
  - > Locks are properly handled (a low priority thread owning a critical resource will be boosted by the thread that requires it)

Use case 1 revisited

- Code executed by a RealtimeThread

```
setPriority(my_RTPriority);
while (true) do {
   compute_data();
   now = System.currentTimeMillis()
   Thread.sleep(next_period - now);
   send_data();
   next_period += period;
}
```

- Problem:
  - > What if a more important RT threads preempts me just before calling sleep ?

RTSJ Benefit 2: Rich Real-Time APIs

- RTSJ provides what you will find in most RT Operating Systems
  - > Absolute Time Clock, Timers, Non Blocking Queues...
- RTSJ provides an additional layer to make your life simpler
  - > Periodic Threads, Asynchronous Event Handlers, RawMemoryAccess, Deadline Miss Monitoring and Management, Asynchronous Transfer of Control...
- RTSJ optionally defines advanced APIs
  - > Cost Overflow Monitoring and Management, Feasibility Analysis...

Use case 1, RTSJ version

- Code executed by a RealtimeThread

```
setPriority(my_RTPriority);
setReleaseParameters(myPeriodParam);
while (true) do {
   compute_data();
   RealtimeThread.waitForNextPeriod();
   send_data();
}
```

  (other variants with Timers, AbsoluteTime wait, ...)

- Problem:
  > Is this sufficient ?

Yes... if optimized for Determinism

- The fastest Garbage Collectors suspend all the threads while they recycle memory
- The fastest JIT compilers make some assumptions to generate more efficient code and must 'deoptimize' threads if the assumption becomes invalid
- Solution 1:
  - RTSJ defines additional threads optimized for determinism and isolated for GC jitter.
  - They are as deterministic as C/C++ code !

NoHeapRealtimeThreads

- Code executed by a NoHeapRealtimeThread

```
setPriority(my_RTPriority);
setReleaseParameters(myPeriodParam);
while (true) do {
   compute_data();
   RealtimeThread.waitForNextPeriod();
   send_data();
}
```

- Problem:
  - > What about memory allocation since 'isolated' from the GC ?

Memory areas for NHRTs

Don't bother if 200 microseconds latencies are OK !!!

- ImmortalMemory for non recycled objects
- ScopedMemory for recycling
  - Memory areas not subject to the Garbage Collector
  - Per area counters to know how many threads are using an area
  - Objects automatically deleted when the count of their area is 0
  - Dynamic read/write checks to guarantee the safety of this recycling

Use case 1, ScopedMemory version

- Code executed by a NoHeapRealtimeThread

```
while (true) do {
   scopedMemory1.enter(runnable);
   // sm1 recycled for the next loop
}
void run() {
   compute_data();
   waitForNextPeriod();
   send_data();
}
```

- Problem:
  - Is send_data() endorsing read/write constraints ?

Agenda

- Background
- Benefits of a Real-Time Java Virtual Machine
- Benefits of Real-Time Garbage Collectors
- Q&A

Making Java Predictable, RTGC

- Real-Time GC
  - Essentially a “garbage collector with knobs”
  - Large interrupts are avoided at the cost of **small, regular, and predictable collections**
  - **The GC runs often enough to recycle memory on time**, depending on allocation and collection rates
- Several models/approaches exist
  - IBM, Aicas, Sun, ...
- Way simpler from a coding point of view
  - Single heap
  - No read/write checks

Overall System Model

Real-Time JVM

Non real-time	• Standard Java Heap > Regular Java threads
Soft real-time	• Standard Java Heap or Scoped or Immortal memory > Real-Time threads > Real-Time Garbage Collector
Hard real-time	• Scoped/Immortal/Heap > NoHeapReal-Time threads > Real-Time threads > Hard Real-Time Garbage Collector

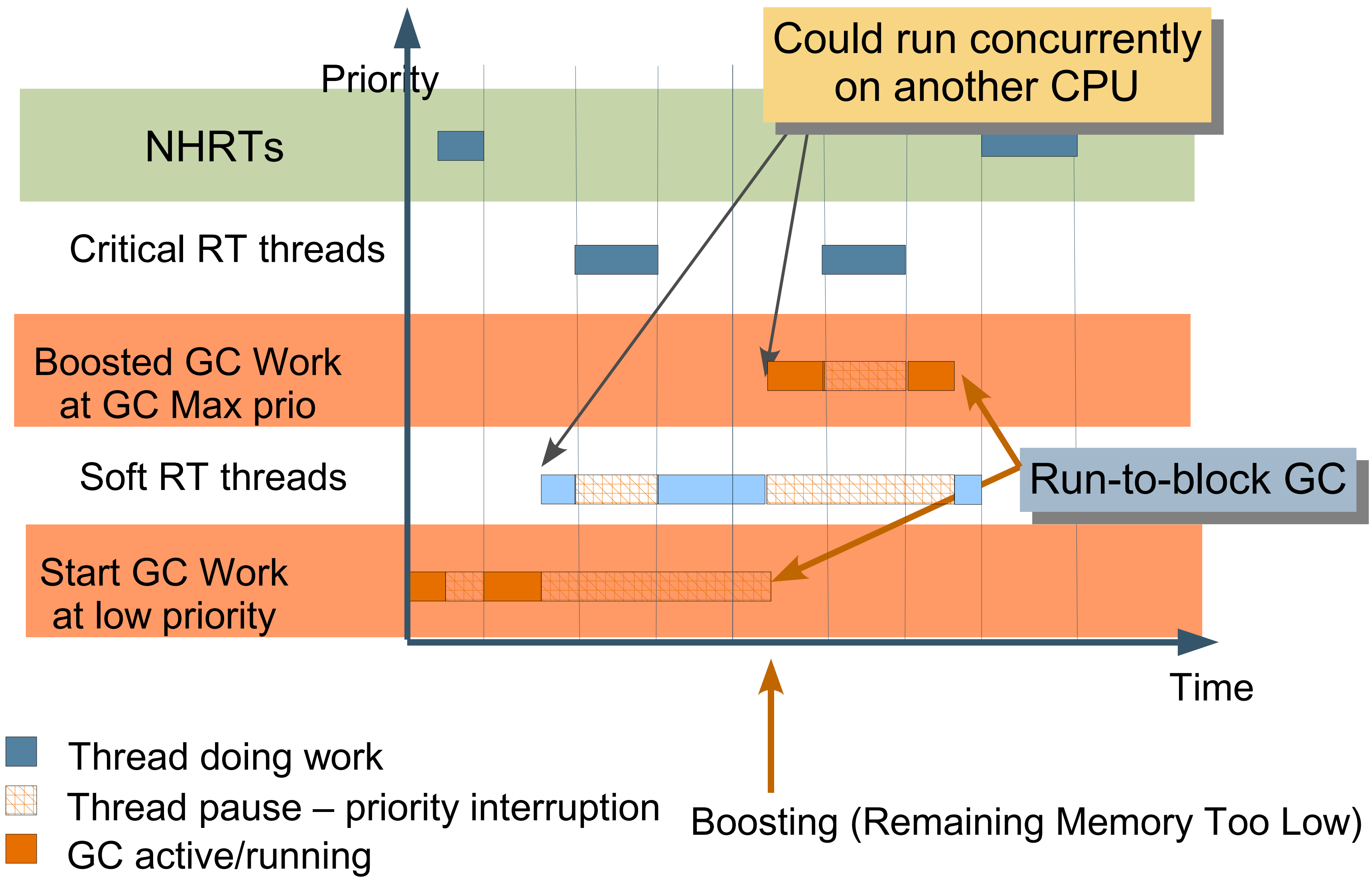
Real-time code and existing, non real-time, code can communicate, share memory, state, and overall environment – something not possible in current real-time solutions

Sun's RTGC

Apply concept of policy to Henriksson's approach

- Goal: Hard RT latencies for high priority GC'd threads
- Solution: implement GC as real-time threads
  - Running at a priority **lower** than the hard RT threads
  - And possibly **concurrently** with the other threads
- Advantages
  - Scalable (no issues with multi-processor support)
  - Flexible

Default RTGC Scheduling Policy



Use case 1 revisited

- Code executed by a **hard** RealtimeThread

```
while (true) do {
   waitForNextPeriod();
   send_data();
}
```

- Code executed by a **soft** RealtimeThread

```
while (true) do {
   compute_data();
   waitForNextPeriod();
}
```

- `softRT.setDeadlineMissHandler(dmh)`

Use case 1 Deadline Miss Handler

- Hard Real-Time AsynchronousEventHandler

```
void handleAsyncEvent() {
    quickly_compute_simpler_data();
    softRTThread.schedulePeriodic();
}
```

or

```
void handleAsyncEvent() {
    softRTThread.setPriority(hardPrio);
    softRTThread.schedulePeriodic();
}
```

Use Case 2 : Events Driven Request with Deadlines

- A request is valid:
  - For a limited time
  - When a set of asynchronous input feeds dependent conditions are true
- Proposed design:
  - Use **a high priority critical thread for the request**
  - If necessary, rely on priority boosting to **safely use locks** to check the <condition,deadline> pair(s)
  - Use AsynchronousEventHandlers to process the input feeds
    - Potentially at different priorities (hard and/or soft)
    - Use Minimum Inter-arrival Time policies to handle overflows

Use Case 3 : NASDAQ
“Reduce I/O Jitter and Context Switches”

- I/O must not hold other requests
- Implemented Solution:
  - One thread per-CPU
    - One front-end to receive requests and executed non I/O ones
    - I/O requests dispatched to another thread
    - I/O completion can be seen as a new request by the front-end
- Benefits from Java RTS deterministic compilation and memory management but :
  - What if I/O requests can be processed at different rates ?
  - Could I handle more non I/O requests ?

Use Case 3 : Why Care about Context Switches ?

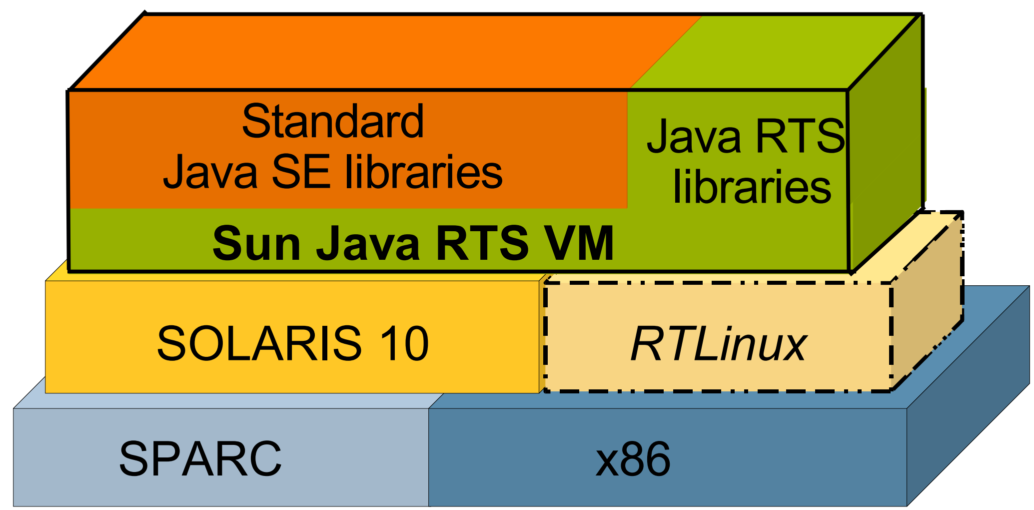
This should not be your role !

- Risk : under-utilization of resources
- Scalable Design:
  - > Chosen number of high priority time-critical front-end threads
    - > Could even be at different priorities depending on the input stream
  - > AsynchronousEventHandler to process I/Os at a lower priority
    - > New I/O processing threads will automatically be created by the VM server thread pool to maximize the number of parallel requests
    - > Requests started later can complete earlier
    - > The server thread automatically grabs a new one... **without context switch**

Sun Java RTS Unique Selling Proposition

Sun's Java Real-Time System – absolute execution predictability with all the benefits of the Java platform

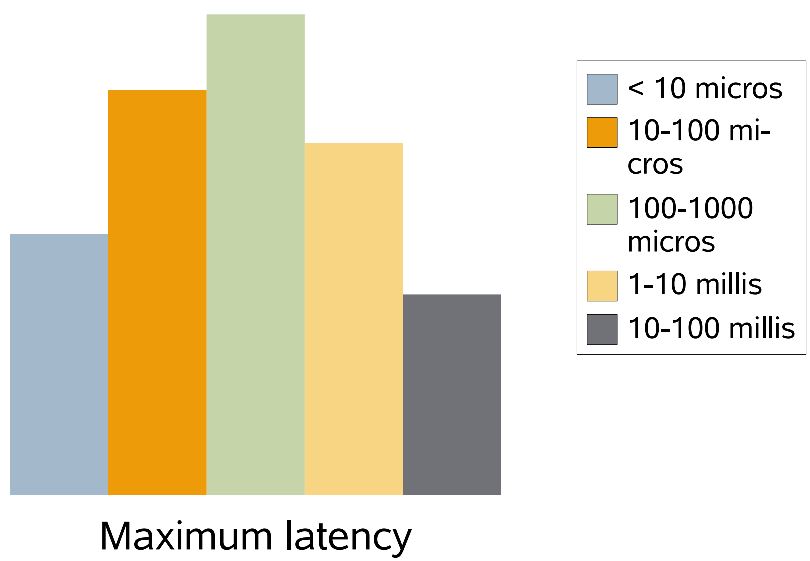
- **Truly predictable, sample latency numbers:**
  - RTGC: 200 microseconds
  - NHRT: 20 microseconds
- **Open**
  - Based on open, community-driven standards



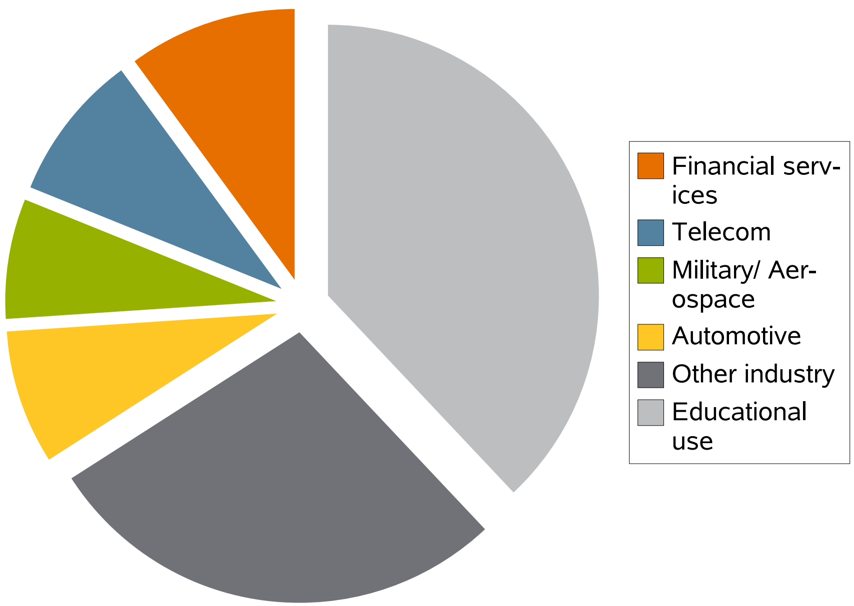
Customer Evaluations

- Over 500 companies and universities from more than 60 countries have evaluated Sun Java Real-Time System
- Some data points from them:

Maximum Latency Required



Evaluations by Industry



Source: Evaluation download survey for Sun Java RTS

Key Benefits of RTSJ

TECHNICAL

- Maximize system utilization
- Cross-platform portability
- Leverage standard OS's
- Real-time tools

BUSINESS

- Eliminate latency outliers
  - Save millions of $ lost due to missed market opportunities, productivity losses etc.
- Eliminate risks inherent in custom and proprietary solutions
- Leverage investment in existing Java code, skills
- Increase developer productivity using Java

Key Benefits of RTSJ + RTGC

TECHNICAL

- Maximize system utilization
- Cross-platform portability
- Leverage standard OS's
- Real-time tools
- Minimize solution complexity
- Better design / architecture choices

BUSINESS

- Eliminate latency outliers
  - Save millions of $ lost due to missed market opportunities, productivity losses etc.
- Eliminate risks inherent in custom and proprietary solutions
- Leverage investment in existing Java code, skills
- Increase developer productivity using Java

Key Benefits of Sun Java RTS

TECHNICAL

- Maximize system utilization even more (hard + soft RT)
- Cross-platform portability
- Leverage standard OS's
- Real-time tools (DTrace)
- Minimize solution complexity
- Better design / architecture choices
- Easy integration of hard, soft, and non time-critical design elements

BUSINESS

- Eliminate latency outliers
  - Save millions of $ lost due to missed market opportunities, productivity losses etc.
- Eliminate risks inherent in custom and proprietary solutions
- Leverage investment in existing Java code, skills
- Increase developer productivity using Java

Questions?

Bertrand.Delsart@Sun.COM