

The Sun Java Real-Time System on Wall Street

Eric J. Bruno
eric.bruno@sun.com

The Sun Java Real-Time System (Java RTS) is the first conformant commercial implementation of Java Specification Request (JSR) -001, the Real-Time Specification for Java (RTSJ). Java RTS enables developers of real-time applications to take full advantage of Java while maintaining the predictability of current real-time development platforms. With Java RTS, real-time components and non-real-time components can coexist and share data on a single system.

When released, Java RTS 2.0 will contain a real-time garbage collector for soft real-time support, along with RTSJ-compliant programming models for hard real-time support. Java RTS 2.0 runs on Solaris 10 systems with either x86/x64 or SPARC processors, and is fully Java 5 compliant; you can run your existing Java application class or JAR files with Java RTS, along with applications built for real-time behavior.

Since late 2006, Sun has been working with many large customers in the financial space (banks, brokerages, exchanges, and so on) to help them discover the benefits of Java RTS in their time-critical financial applications. Before we get into the specifics of Java RTS in the financial space, let's discuss the concept of real-time.

What is Real-time?

In computer science, real-time computing (RTC) is the study of hardware and software systems which are subject to a "real-time constraint," such as operational deadlines for event to system response. By contrast, a non-real-time system is one for which there is no deadline, even if fast response or high performance is desired or even preferred. The needs of real-time software are often addressed in the context of real-time operating systems, and synchronous programming languages, which provide frameworks on which to build real-time application software.

A distinction can be made between those systems which yield incorrect, or undefined, results if time constraints are violated (hard or immediate real-time), and those which will not (soft real-time). A system is said to be hard real-time if the correctness of an operation depends not only upon the logical correctness of the operation but also upon the time at which it is performed. It is often misconstrued that a system has a hard real-time requirement if its deadline is small. This is incorrect; the magnitude of the deadlines do not define a hard or soft real-time system. A hard real-time requirement is any requirement that states the system will go to an abnormal state if a deadline is missed.

Soft real-time systems are typically those used where there is some issue of concurrent access and the need to keep a number of connected systems up to date with changing situations. In soft real-time scenarios, jitter can be tolerated, but predictability is still a requirement. In a nutshell, hard real-time systems effectively require a bounded response to an event, whereas soft real-time systems can tolerate missed deadlines.

The overall goal is to ensure that a system performs its tasks, in response to a real-world event, before a measurable deadline. Regardless of whether that deadline is measured in microseconds, or days, as long as the task completes before that deadline, the system is real-time. In other words, the time delay from when a real-world event occurs (such as an object passing over a sensor, or the arrival of a stock

market data feed tick) to the time your code completes its processing should be bounded. The ability to meet this deadline must be predictable and guaranteed, all the time, in order to provide the determinism needed for a real-time system.

Real-time vs. Real Fast

The term, *real-time*, is used in many contexts in the computer field, often incorrectly. For instance, real-time is often used to describe a system that is considered to have fast performance. However, the ability to react quickly to external events does not accurately describe a real-time system. Other times, the term is used to describe a system that responds to events as they occur (as opposed to discovering an event by checking for it at a later time). An example of an application that is incorrectly labeled real-time is a typical instant messaging system.

Although an instant messaging system allows you to communicate with someone in a quasi real-time manner (as opposed to an email system), this is still not a true real-time application. The delay of a message by varying degrees of latency does not affect the correctness of the system. As long as a message in an instant messaging system is delivered within a reasonable time window (many seconds, up to a minute), the system is behaving as expected.

Real-time and Throughput

Another area of system performance that is often confused with real-time is that of system throughput. Throughput is often used to describe the number of requests, events, or other operations, that a software system can process in any given time frame. You often hear of software positively characterized with terms like “messages-per-second,” or “requests-per-second.” A system with high throughput can give its operators a false sense of security when used in a real-time context.

This is because a system with high throughput is not necessarily a real-time system; although it is often misunderstood to be. For instance, a system that supports thousands of requests-per-second may have some requests that take up to a second to be responded to. Even though a majority of the requests may be handled with low latency, the existence of some messages with large latency represents outliers (those outside the normal response time). Because the degree of, and the amount of, these outliers is unpredictable, these high-throughput systems are not real-time systems.

Instead, whereas the objective of fast computing is to minimize the average response time of a given set of tasks, the objective of real-time computing is to meet the individual time-critical requirement of each task. Consider this anecdote: there once was a man who drown in a river with an *average* depth of 6 inches [Buttazzo05]. A real-time system is characterized by its deadline (the maximum time within which it must complete its execution).

The Java Real-time System, version 2.0

Standard Java has some deficiencies that make it less than ideal in the real-time world. For instance, the garbage collector (an internal JVM thread that reclaims old objects to free space on the heap) can run at any time, for any length of time, thereby introducing latency and non-determinism to your application. However, the garbage collector is not the only culprit in a standard Java application. The just-in-time (JIT) compiler, which optimizes your Java code for the machine it's running on, can be invoked at any time, even for code that has previously been optimized (JIT-compiled).

Further, Java SE does not typically honor thread priorities as specified in your code. For instance, a Thread that you create and specify to run at highest priority will likely not run at the highest priority in

the system. Therefore, other application threads can preempt your Java application at any time, leading to further non-determinism.

The Java Real-time System (Java RTS) from Sun is meant to resolve these issues. Java RTS is an RTSJ-compliant Java virtual machine with the following characteristics:

- It is 100% Java 5 compliant
- It supports applications with both hard and soft real-time requirements
- It exposes true high-resolution timers
- It implements true thread priorities (the highest priority real-time thread is truly the highest priority thread in the entire system)
- It does not force you to modify your existing code in unreasonable ways
- It does not change the syntax of the Java language
- It offers tools support to aid in development, debugging, and deployment of real-time Java applications

Java RTS currently runs only on Solaris 10 because Solaris is a true real-time operating system. Solaris has contained special kernel features to support real-time processes and threads almost since its inception. Solaris supports the following real-time features:

- High resolution timers
- Pre-emptive scheduling
- Built-in support for real-time processes
- A special class of thread priorities for real-time threads
- Separate kernel dispatch queues for real-time threads
- Run-to-block real-time thread behavior
- Deterministic, bounded, and guaranteed dispatch latency
- Priority inheritance
- Processor sets
- Interrupt shielding

The last two items on the list above deserve explanation. Solaris allows you to define processor sets, which are sub-sets of processing cores that you group as a set. For example, in a four-core system, you can define a processor set that contains two cores, and then run your Java RTS application on that set. From that point onward, the two cores in the processor set will run only the Java RTS application you specified. It's sort of like dedicating a portion of the overall system to your application.

What's more, with interrupt shielding, you can tell Solaris to shield the cores in your processor set from handling interrupts. The result is that the two cores will be dedicated to running your Java RTS application, and will not be interrupted by any other system thread, including low-level system interrupts.

Java RTS and Financial Systems

Companies in the financial space, with systems that must respond to financial market events, have a

need for real-time system behavior. Financial markets tend to move at a fast, steady, pace. Even when the pace isn't fast, it's still crucial to respond to market events and trends within a bounded time-frame in order to make money, or to avoid losing it. For example, every millisecond of latency in a financial application represents a potential loss of \$100 million.

For companies that provide data to financial customers, it's critical to deliver this data within a bounded time frame. Therefore, both the data receivers (i.e. trading systems), and the data generators (i.e. market data feeds) consist of time-critical software systems that have real-time requirements. Unbounded latencies introduced by the Java VM garbage collector, or other activities, represents significant financial loss to the companies operating these systems. Inversely, the elimination of these latencies, and the added promise of determinism, represent a significant competitive advantage. Java RTS is the platform that delivers real-time Java application behavior.

To prove the benefit of Java RTS over standard Java in the real-time space, we've built a simulated limit/stop order trading system that works with both standard Java, and Java RTS. The results are stunning and impressive. Let's take a moment to examine the concepts of limit and stop orders.

What is a Limit Order?

A limit order is an order to buy a security at no more (or sell at no less) than a specific price. This gives the customer some control over the price at which the trade is executed.

A *buy limit order* can only be executed by the broker at the limit price or lower. For example, if an investor wants to buy a stock but doesn't want to end up paying more than \$20 for the stock, the investor can place a limit order to buy the stock at any price up to \$20. By entering a limit order rather than a market order, the investor will not be caught buying the stock at \$30 if the price rises sharply.

A *sell limit order* can only be executed at the limit price or higher.

A limit order may never be executed if the market price surpasses the limit before the order can be filled. Because of the added complexity, some brokerages will charge more for executing a limit order than they would for a market order.

What is a Stop Order?

A stop order (sometimes known as a *stop loss order*) is an order to buy or sell a security once the price of the security reaches a specified price, known as the *stop price*. When the specified price is reached, the stop order is entered as a market order. Stop orders are used to try to limit an investor's exposure in the market.

A *sell stop order* is an instruction to sell at the best available price after the price goes below the stop price. A sell stop price is always below the current market price. For example, if an investor holds a stock currently valued at \$50 and is worried that the value may drop, he/she can place a sell stop order with the broker at \$40. If the share price drops to \$40 for whatever reason, the broker will sell the stock at the next available price. This can limit the investor's losses (if the stop price is at or below the purchase price) or lock in at least some of the investor's profits (if the value of the security has risen between when the security was purchased and the stop order placed).

A *buy stop order* is typically used to limit a loss (or to protect an existing profit) on a short sale. A buy stop price is always above the current market price. For example, if an investor sells a stock short (borrows stock and sells it immediately at current market price (Shorting), and the investor hopes the stock price goes down in order to give the borrowed shares back at a lower price (Covering) while pocketing the difference), the investor may try to protect himself against losses if the price goes too

high using a buy stop order.

With a stop order, the customer does not have to actively monitor how a stock is performing. However because the order is triggered automatically when the stop price is reached, the stop price could be activated by a short-term fluctuation in a security's price. Once the stop price is reached, the stop order becomes a market order. In a fast-moving market, the price at which the trade is executed may be much different from the stop price.

The use of stop orders is much more frequent for stocks, and futures, that trade on an exchange than in the over-the-counter (OTC) market.

The Java RTS Financial Demonstration Application

The demonstration limit/stop order trading system is built with two main components; a data feed application, and a trading engine application. The trading engine executes two main threads; one that listens to the data feed and updates the current market price for stocks it tracks, and another that runs a time-critical loop that compares existing limit/stop orders against the current market values. When the market price of an equity moves within range of a limit or stop order, the trading engine executes the trade.

For the non-real-time implementation of the trading engine, it's conceivable that an outside event (such as garbage collection) can interrupt the time-critical loop, and result in unbounded latency. In that time frame, the market may move within range, and then out of range of one or more of the open orders in the system. In fact, this is exactly what happens, and demonstration shows it clearly

In this scenario, the missed opportunity represents financial loss to the operating institution since it will still be required to honor the affected orders with its customers, even though the ideal trading window was missed. Although the chance of garbage collection affecting trades is marginal, it represents significant risk, and as this demo shows, that risk is real.

However, when the same trading engine (with minor changes to the threading model) runs with Java RTS, the Java VM is able to honor the deadlines within the time-critical code, and no trade opportunities are missed. All market ticks are captured, and all trading opportunities are responded to. The result is a system that can reliably respond to market events, and trade customer orders, when the right conditions occur, at the precise moments in time that they occur.

If you're interested in the Java Real-Time System, and if you'd like to see the demonstration financial application in action, contact your local Sun sales team.

References

[Buttazzo05] Giorgio C. Buttazzo, *Hard Real-Time Computing Systems – Predictable Scheduling Algorithms and Applications*, Springer Science and Business Media, 2005