Real-time Extensions to the Java™ Platform
A Progress Report

William Foote
Sun Microsystems
Real-Time Java™ VM Group
bill.foote@eng.sun.com

Abstract

Java™ technologies have had a major impact on the software industry. The Java™ Platform has brought a simple, powerful object-oriented language and set of libraries to a large developer community. It has delivered on its promise to bring increased productivity, "write once run anywhere," security, and dynamically downloadable code to desktop and workstation environments.

Java technologies are now being extended into embedded devices [1]. These devices often require software that exhibits realtime characteristics. Java language semantics are not specified tightly enough for such software, and certain API extensions are required. This paper explores the technical issues involved in defining real-time extensions to the Java platform that meet this need.

1. Introduction

Sun Microsystems is actively collaborating with members of the industry interested in real-time functionality to develop a set of standard extensions to the Java™ Platform. With these extensions, we will bring the high productivity, reliability, portability and dynamic features of Java™ technology to real-time programming. The specification for these extensions will be developed through Sun's open process, and will involve expertise and feedback from a wide range of sources.

An important source of requirements will be the requirements developed by the NIST Requirements Working Group for Real-time Extensions [2]. Their web site features an archive of the e-mail alias used by group members. Participation in the group is completely open to the public, and it has been a very valuable forum. Sun is a member, and we have contributed to the requirements document.

To help advance debate, we have posted a proposed specification at the NIST site. This specification is meant only as a starting point for discussion. When the expert group is formed under Sun's open process, this document will be one of the inputs. It is probable that the final specification will differ significantly from this preliminary proposal, however. This preliminary document [3] is available at the NIST web site.

The remainder of this paper will highlight some of the more interesting technical issues that have come up, and indicate some possible opinions on the direction that a first version of a real-time extension specification might take.

2. Real-time Issues

2.1 Negotiating Components

One of the interesting desirata to be discussed int the NIST group is the idea of a Negotiating Component. The goal is to have a component that negotiates with the runtime about resource requirements. This would allow a new negotiating component to be dynamically introduced into a running system. If sufficient resources are not available, the new component could be rejected, or its resource requirements could be negotiated down via some API. If a suitable resource contract could be negotiated, then the new negotiating component would be executed with a guarantee that the resource requirements of the new component, and all others in the system, would be met.

This is an ambitious goal. A solution to this problem would advance the state of the art of real-time programming. Finding a solution that preserves compatibility with existing Java runtime semantics adds interesting technical challenges. Because of the relatively high risk inherent in the development of a new real-time programming paradigm, Sun believes that a negotiating component should not be a part of an initial "low-level" real-time extension. We would enthusiastically support the parallel development of such an API, in the form of a third-party API, or as a "high-level" specification developed through Sun's open process.

2.2 Thread Scheduling

Two major thread scheduling paradigms have been discussed in the NIST group: priority-based scheduling, and deadline-based scheduling.

Sun's preliminary proposal features a javax.realtime.RealtimeThread with tightly specified service guarantees. A RealtimeThread is priority based, and it has a large number of priority levels (128 in the current proposal). RealtimeThreads are fixed-priority, in the sense that the scheduler will not adjust thread priorities in a timesharing manner. They are preemptively scheduled, that is, if two RealtimeThreads are schedulable and have different effective priorities, the one with the higher effective priority will always be scheduled. RealtimeThreads feature a priority inheritance protocol.

Priority-based scheduling is a well-established technique that is present in nearly all commercial real-time operating systems. A different paradigm, deadline-based scheduling, is of considerable interest to the research community. Deadline-based scheduling enables a programmer to write a task in terms of its requirements, without resorting to global knowledge about other tasks in the system. It's interesting to note that negotiating components would probably require some kind of deadline-based scheduling.

2.3 Priority Inversion

The problem of priority inversion, where a high priority task waits on a lock held by a lower-priority task, is an area where the execution semantics of Java need tighter specification. Two solutions have been discussed: priority inheritance, and priority ceiling.

Sun's preliminary proposal features priority inheritance, both for synchronized code, and for explicit ReentrantMutex instances. If a real-time thread is blocked on a lock held by a lower priority thread, the effective priority of the thread holding the lock is boosted to that of the blocked thread. This can happen recursively, that is, if the thread holding the lock is itself blocked, another thread could have its effective priority boosted.

A related problem is the wait queue associated with a mutex or a synchronization lock. In standard Java, there is no guarantee about the order in which threads are removed from the wait queue associated with a lock. A real-time extension will need to specify this behavior more tightly, so that threads are removed in priority order.

2.4 Asynchronous Event Handling

Asynchronous events can be broadly characterized into two groups: External (hardware) interrupts, and software signals. The Java Platform does not have a facility for dealing with asynchronous events. The old mechanism of asynchronously sending an exception to a thread via Thread.stop(Throwable) has been deprecated, and is not a suitable mechanism for real-time programming. The reasons behind the deprecation has been published in a rationale [4].

Hardware interrupts usually target the machine as a whole, that is, they don't naturally have any specific thread as their target. In Sun's preliminary proposal, they are addressed with the classes RealtimeEvent, RealtimeEventGenerator and their subclasses. A RealtimeEvent can represent asynchronous IO, or a timer firing. Each event is generated with an associated priority, which corresponds to real-time thread priorities. The event is implicitly processed by a system thread at the indicated thread priority.

Software "interrupts" offer a greater challenge, because they do target a particular thread. Safely terminating a thread in an asynchronous manner is highly problematic. Software interrupts can be very useful, however. The requirements working group has identified the following reasons why they are desirable:

- Timeouts
- Refinement Computations
- Change in execution environment
- Because polling requires global knowledge
- Shutdown (for fail safes)
- Overload situations
- Would help support Ada

For asynchronous thread termination to be safe, it must be possible to defer it during critical sections. The requirements working group has identified the following places where asynchronous events should be deferred:

- In a finally clause
- In code that has been "marked" by the user

Asynchronous exceptions would not be deferred in a catch clause, in synchronized code, or in a finalizer.

One advantage of adopting some kind of software interrupt is that this more general model could subsume hardware interrupt. In this case, there's an additional performance benefit: Delivery of a hardware interrupt to a user thread would not involve a context switch from the system interrupt-delivery thread.

2.5 Garbage Collection

Java programs tend to use the heap extensively, and there is no provision for explicit deallocation of heap-allocated objects. Providing a garbage collector that offers the needed real-time guarantees is one of the key technical challenges that must be addressed.

The NIST requirements working group defines real-time garbage collection as follows:

> 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.

Finalizers present a particular challenge. The Java Language allows arbitrary finalization code to be associated with any class, by overriding the finalize() method. Before the storage associated with an instance of such a class can be reclaimed, the system must run the finalizer. Because that code can be arbitrarily time consuming, the system cannot guarantee when objects with non-trivial finalizers can actually be reclaimed. This interferes with a promised guaranteed reclamation rate of a garbage collector.

The real-time extension will need to address finalization in some way. This may include a tighter specification of when and at what thread priority finalizers are run. It may also include rules about under what conditions it is safe to use objects with non-trivial finalizers.

2.6 Stack Allocation

The concept of stack allocation of object values, and the related issue of pass-by-value semantics of method arguments was the subject of a spirited discussion in the NIST group. Stack allocation of object types is desirable, because the execution time properties of stack allocation are more easily understood than heap allocation. In addition, stack allocation is generally much faster.

Adding stack allocation to the Java platform is highly problematic. Java offers a strong guarantee that dangling references will never happen. This guarantee is incompatible with the idea of passing references to stack-allocated objects as method parameters, because in general it is impossible to guarantee at verification time that the method will not create an alias to that object that outlives the stack frame in which it was created. If stack-based objects cannot be passed by reference, then it is difficult to do interesting things with them.

The Sather Language [5] has an interesting solution: value types. This is intriguing, but probably too major a change to the Java Language for a real-time extension.

With some regret, the idea of adding stack allocation of object types has been shelved for now.

3. Conclusion

Extending the Java Platform to make it suitable for real-time programming presents interesting technical and API development challenges. Some of these challenges are well-understood, and solved in many commercial real-time operating systems. Other challenges are less well understood, and a solution to them would advance the state of the art.

The less well-understood areas include resource negotiation by "negotiating components," deadline-based scheduling, and certain aspects of asynchronous event handling. It is unclear to what extent it will be possible to address these difficult issues in a first real-time extension specification. Real-time garbage collection is a difficult problem, but a solution is central to any real-time environment in which traditional Java programming is possible. More straightforward problems include thread scheduling, priority boosting, and the asynchronous handling of external events.

Sun is committed to producing high-quality standard extensions though our open process. We strongly believe that the best standard will be achieved through a participatory process that takes ideas from many sources. Your feedback and insights, now and in the future, are welcomed and appreciated.

References

[1] Sun Microsystems, EmbeddedJava™ Home Page, http://java.sun.com/products/embeddedjava/

[2] Lisa Carnahan, NIST Requirements Working Group Homepage, http://www.nist.gov/rt-java/

[3] Nedim Fresko, William Foote, "Proposed Java™ Real-Time Extension Specification," http://www.sdct.itl.nist.gov/~carnahan/real-time/sun/index.html

[4] Sun Microsystems, JDK 1.2 documentation, "Why are Thread.stop, Thread.suspend and Thread.resume Deprecated?" http://java.sun.com/products/jdk/1.2/docs/guide/misc/threadPrimitiveDeprecation.html

[5] The International Computer Science Institute, "Sather object oriented programming," http://www.icsi.berkeley.edu/ sather/