Technical overview
White paper

WebSphere® software

Creating predictable-performance Java applications in real time.

By Michael Dawson, Mike Fulton, Greg Porpora, Ryan Sciampacone and Mark Stoodley, IBM Software Group, and Vernon Mauery, IBM Systems and Technology Group

March 2007

Contents

Executive summary

This white paper provides a short primer on real-time applications and the issues and concerns with using a standard Java™ Virtual Machine (JVM) to run them. It then describes how IBM has addressed most of these problems in the IBM WebSphere® Real Time product. In particular, this white paper discusses the innovations made in each of the core components of the new real-time JVM, including the Metronome garbage collector, the J9 JVM, IBM ahead-of-time (AOT) and just-in-time (JIT) compilers, the extensions to IBM's core class libraries and the new class libraries provided as part of IBM Real-Time Specification for Java (RTSJ) support.

What is a real-time application?

Real-time is a particularly broad term that is used to describe applications that have real-world timing requirements. For example, a sluggish user interface does not satisfy the generic real-time requirements of an average user. This form of application is often described as a *soft* real-time application, because no harm comes from the application being slow to respond, other than loss of sales for a poor product. The same requirement might be more explicitly phrased as "The application should not take more than a quarter of a second to respond to a mouse click." If the requirement is not met, it is a soft failure—the application can continue and the user, although unhappy, can still use the application. In contrast, applications where real-world timing requirements must be strictly met are typically called *hard* real-time applications. An application controlling the rudder of an airplane, for example, cannot be delayed for any reason because the result would be catastrophic.

A key aspect of real-time requirements is response time. When writing hard or soft real-time applications, it is critical to understand the response-time constraint. The techniques required to meet a hard one-microsecond response are significantly different than those required for a hard 100-millisecond response. In practice, achieving response times below tens of microseconds requires a combination of custom hardware and software, possibly with no operating system.

This white paper describes the WebSphere Real Time product, which can provide hard response-time guarantees for real-time Java applications requiring responses of tens of microseconds and more.

Can Java technology be used for real-time applications?

Standard Java applications running on a general-purpose JVM, on a general-purpose operating system, such as Microsoft® Windows® or Linux®, can only hope to achieve soft real-time requirements in the hundreds of milliseconds. Several fundamental aspects of the language are responsible, including class loading and compilation, garbage collection, and thread management. Some of these issues can be mitigated, but only with significant work.

Class loading and compilation

A JVM must delay loading a class until a program first refers to it. This class loading can take a variable amount of time depending on the speed of the medium from which the class is loaded, the size of the class and the overhead incurred by the class loaders themselves. The delay to load a class can often be as high as 10 milliseconds. If you need to load tens, hundreds or thousands of these classes, the loading time itself can cause a significant and possibly unexpected delay. You can use careful application design to load all classes at application startup, but class loading must be performed manually because the Java language specification[1] does not permit the JVM to perform this step early.

Compiling Java code to native code introduces a similar problem. Most modern JVMs interpret Java methods, compiling only frequently run methods as required to native code. Not compiling immediately results in fast startup and helps reduce the amount of compilation that needs to be performed, but it creates a problem for a hard real-time application because methods run more slowly when the interpreting phase is initially running. As with class loading, using the compiler class to programmatically compile methods at application startup can mitigate this problem, but maintaining such a list of methods is tedious and error prone.

Garbage collection

Another source of frustration for hard real-time programmers using Java is garbage collection. Errors introduced by the need to explicitly manage memory in languages such as C and C++ are some of the most difficult problems to diagnose. Proving the absence of such errors when an application is deployed is also a fundamental challenge. One of the major strengths of the Java programming model is that the JVM, not the application, handles memory management, which helps eliminate this burden for the application programmer. Unfortunately, traditional garbage collectors can incur very large application delays that are virtually impossible for the application programmer to predict. Delays of several hundred milliseconds are not unusual. One way to solve this problem is to prevent garbage collections by creating a set of objects that are reused, helping to ensure that the Java heap memory is never exhausted. In practice, this approach generally fails because it prevents programmers from using many of the class libraries provided in the Java Development Kit (JDK) and by other class vendors, which typically create many temporary objects.

Thread management

Standard Java does not provide any guarantees for thread scheduling or thread priorities. An application that must respond to events in a well-defined time has no way to ensure that another low-priority thread won't get scheduled in front of a high-priority thread. To compensate, a programmer would have to partition an application into a set of applications that can then be run at different priorities by the operating system. This approach would increase the overhead of these events and make communication between the events far more challenging.

RTSJ: Addressing the challenges of real-time environments

RTSJ was created to address some of the limitations of the Java language and provide solutions to some of the problems outlined in the previous section. The RTSJ addresses several areas including scheduling, threading, memory management, synchronization, time and clocks, and asynchrony.

Scheduling

Real-time systems need to control how threads will be scheduled and guarantee that, given the same conditions, threads are scheduled in a predictable way. Although the Java Class Library (JCL) includes the concept of thread priorities, the JVM is not required to enforce priorities. In addition, non-real-time Java implementations typically use a round-robin preemptive scheduling approach with unpredictable scheduling order. With the RTSJ, true priorities and a fixed-priority preemptive scheduler with priority inheritance support is required for real-time threads. This scheduling approach helps in that the highest-priority thread can always be the one running, and it will continue to run until it releases the processor voluntarily or is preempted by a higher-priority thread. Priority inheritance helps ensure that priority inversion is avoided when a higher-priority thread needs a resource held by a lower-priority thread.

Threads

The RTSJ adds support for two new thread classes: RealtimeThreads and NoHeapRealtimeThreads (NHRTs). These new thread classes provide support for priorities, periodic behavior, deadlines with handlers than can be triggered when the deadline is exceeded, and the use of memory areas other than the heap. NHRTs cannot access the heap, and so, unlike other types of threads, NHRTs do not need to be interrupted or preempted by garbage collection. Real-time systems typically use NHRTs with high priorities for tasks with the tightest latency requirements, RealtimeThreads for tasks with latency requirements that can be accommodated by a garbage collector and regular Java threads for everything else.

Memory management

Although many real-time systems can tolerate the small delays resulting from a deterministic garbage collector, there are cases where even these delays are not acceptable. The RTSJ defines immortal- and scoped-memory areas to supplement the standard Java heap. Objects allocated in the immortal-memory area are accessible to all threads and are never collected, representing a limited resource to use carefully. Scoped-memory areas can be created and destroyed under programmer control. Each scoped-memory area is allocated with a maximum size and can be used for object allocation. To help ensure the integrity of references between objects, rules govern how objects in one memory area (heap, scope or immortal) can refer to objects in another memory area. More rules define when the objects in a scope are finalized and when the memory area can be reused. Because of these complexities, the use of immortal and scoped memory should be limited to components that cannot tolerate garbage-collection pauses.

Synchronization

Synchronization must be carefully managed within a real-time system to help prevent high-priority threads from waiting for lower-priority threads. The RTSJ includes priority inheritance support to manage synchronization when it occurs, and provides the ability for threads to communicate without synchronization using wait-free read and write queues.

Time and clocks

Real-time systems need higher-resolution clocks than those provided by standard Java. The new HighResolutionTime and Clock classes encapsulate these time services.

Asynchrony

Real-time systems often manage and respond to asynchronous events. The RTSJ includes support for handling asynchronous events triggered by a number of sources including timers, operating system signals, missed deadlines and other application-defined events.

WebSphere Real Time: A robust tool for managing real-time environmnents

The WebSphere Real Time product provides a conformant RTSJ implementation with several enhancements to enable standard Java code to be used in a broader spectrum of real-time environments. Many of the problems faced by developers of real-time systems are addressed by the WebSphere Real Time product and associated tools available from the IBM alphaWorks® Web site at www.alphaworks.ibm.com/tech. Figure 1 outlines the WebSphere Real Time product architecture.

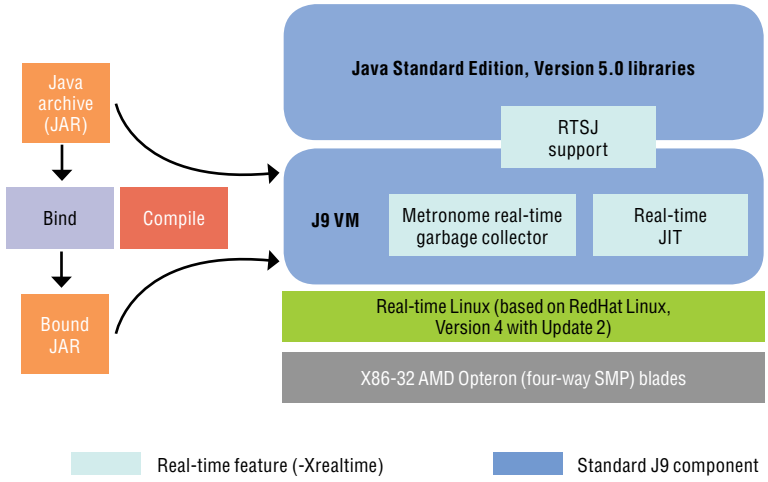

Figure 1. WebSphere Real Time architecture

Real-time Linux

The real-time Linux kernel is created from the mainline Linux kernel with some patches applied to help reduce latency for real-time applications and improve kernel performance. The patches address many of the real-time programming issues such as timing, interrupt latency, task scheduling and kernel preemption. Some of the major advances the real-time Linux kernel has made in helping to reduce latency are discussed in this section.

High-resolution time and timers

Real-time Linux introduced two very important parts that help reduce latency and provide higher precision for timers. First, timers needed a much higher-precision data type to work with, so a 64-bit type, ktime_t, was created. The ktime_t data type is architecture-dependent and allows for representing times with nanosecond precision. Complementing this data type, a new set of functions was created to manipulate the times in the most efficient manner for each architecture. Second, a more-efficient timer expiration mechanism was developed to provide new clock-events code for programmable event interrupts and more-efficient timer-sorting algorithms. Rather than having the timers expire at the low-resolution system tick, timers can now independently expire using a high-resolution timer, enabling them to expire within a few microseconds of each other. Then a binary tree and a sorted list are used for more-efficient, time-bounded timer organization. This new infrastructure along with the ktime_t data type allows for low-latency timer operations at a higher resolution than before.

Fully preemptible kernel

The mainline kernel has three preemption models—no forced preemption, voluntary kernel preemption and preemptible kernel. Real-time Linux offers one more option, complete preemption. Although the mainline kernel preemption choices do offer the ability to preempt some parts of the kernel, many spin-locks still cannot be preempted. Real-time Linux replaces most of the spin-locks with mutexes so that there are fewer places that the kernel cannot be preempted. Because it is legal to sleep (be preempted) when holding a mutex, this capability opens up nearly all of the running kernel paths to preemption so that the kernel preemption can occur in only a few places, and all these critical sections are deterministic and short.

Interrupt handlers are another point of latency caused by the lack of preemption. To minimize latency, real-time Linux allows a real-time process to preempt interrupt handling by converting interrupt handlers into real-time kernel threads. This capability enables them to be scheduled, preempted and prioritized just like any other process. Thus, the only non-preemptible portion of interrupt handling is the few instructions that run in interrupt context to mark the interrupt handler thread as runnable.

Symmetric multiprocessing (SMP) real-time scheduling

Real-time systems require a strictly deterministic scheduling algorithm to run properly. On a single-processor system, performing this task is fairly simple – you just look at the run queue and select the task with the highest priority. Multiple processors turn this simple routine into a complex puzzle. To avoid lock contention between processors, each processor has its own run queue. Periodically, the run queues get balanced to maximize processor utilization. With non-real-time tasks, this procedure is acceptable because eventually, processor-bound tasks run out of processor time and are removed from the queue, and every process gets the chance to run. But this process does not work with real-time tasks. If one processor has two real-time tasks in its queue, one of them is running and the other waiting. A second processor might be running the highest-priority task in its queue, but that task is not necessarily a higher priority than the second task in the first-run queue. With the real-time kernel, if a processor has more than one real-time task, the priority of the real-time tasks on every processor must be compared to determine which tasks should be running. In this way, deterministic scheduling is preserved across multiple processors.

Priority inheritance

Priority inheritance is the real-time kernel response to priority inversions. In the case where a low-priority process holds a lock that a high-priority process is blocked on, it is possible to indefinitely delay both the low- and high-priority processes with a processor-intensive, medium-priority process. The real-time kernel doesn't try to detect the priority inversions. Instead, it avoids priority inversions by raising the priority of the process that owns the lock to be the same as that of the highest-priority process that is being blocked on that particular lock, until the process relinquishes the lock. In this manner, blocked high-priority processes are delayed no longer than absolutely necessary. The kernel uses priority-inheritance mutexes internally to avoid priority inversions inside the kernel.

Fast user-space mutexes (futexes)

Futexes were created to help reduce overhead on mutexes as much as possible. A *futex* is a fast user-space mutex, because it only needs kernel intervention in the case of lock contention. A process does an atomic value exchange to replace the value of the futex with its process ID. If it reads a zero from the exchange, it owns the mutex. Otherwise, it jumps into kernel space to get put on a wait queue. By dividing the lock path into a slow path and a fast path, the common case of claiming an unlocked mutex becomes even faster, with little extra overhead in the slow path.

In addition to creating a mutex that can be locked in user-space, real-time Linux also has the notion of robust mutexes. This function means that when a process holding a lock is terminated, other processes blocked on the lock can recover it in an effective manner, enabling better failure recovery in real-time tasks.

Real-time garbage collection: Metronome

Garbage collection in Java shifts the burden of memory management from the application developer to the JVM. Although the act of reclaiming storage is transparent to the application, it can be visible from an application's performance and behavior. Unpredictable garbage-collection pauses can occur while running a program, and these sometimes lengthy pause times make classic JVMs unsuitable for the real-time market. Although the RTSJ[2] provides capabilities to circumvent these garbage-collection pause times, it does so at the cost of requiring programmers to do their own Java memory management.

Garbage collection typically consists of stopping the running of the Java program, tracing through all live objects in the system and then reclaiming the storage of dead objects. This method of garbage collection is known as *stop-the-world (STW) garbage collection*. The efficiency of the algorithm and type of work being done, including compacting memory to help reduce fragmentation, can contribute to the size of the application's pause. Many modern garbage-collection tactics break this STW pause into more-manageable pieces, either operating as a series of STW increments to achieve a single garbage-collection cycle, or running concurrently with the active program, exacting a tax to the application and running threads to progress through a garbage-collection cycle.

Real-time tasks require an environment in which they can meet deadlines in a specified period of time; if the deadline cannot be met in that time (interruptions occur because of garbage collection, for example), then the real-time guarantees have failed. Although garbage-collection-induced pauses are allowed, they cannot cause the task to miss its deadline, and so there needs to be a balance of how much the garbage collector can pause the task compared to how much processing time the task can receive. Typical solutions involve reducing the times of garbage-collection pauses through a variety of means (such as concurrency, increments and performance improvements).

Reducing pause times to a guaranteed maximum is not enough to achieve real-time performance levels. Consider situations in which two or more garbage-collection pause points occur very close together in time; although the pause times of each might be small, the total garbage-collection pause for a time interval can actually become quite high. What are actually needed are both a low pause-time guarantee from the garbage collector, as well as a guarantee that the use of a program not be lower than a certain percentage during a specified window of time.

The ratio of time spent in the application over a given window of time is known as *utilization*. The units of measurement for utilization allow an application developer to determine if the real-time task requirements can be achieved given a particular utilization in a system. These tasks are typically measured over the course of a window of time; pause times that require tighter timing requirements are encouraged to use the RTSJ. The Metronome garbage collector achieves this capability by providing low individual pause increments in the garbage-collection cycle, as well as targeting a utilization rate over a window of time.

Achieving utilization rates

The Metronome garbage collector is an incremental collector that effectively divides an STW collector into a series of short increments within which it accomplishes a garbage collection. These increments are short (~500 microseconds) and are scheduled so that the target utilization (defaulting to 70 percent) is met over the set window of time (10 milliseconds). This capability is in contrast to other soft real-time offerings, whose pause times can reach 50 milliseconds or more in significantly larger windows of time.

The Metronome garbage collector uses a time-based method of scheduling, which interleaves the collector and the mutator (application) on a fixed schedule.[3] Time-based scheduling was chosen because allocation rates in a program are uneven; scheduling work relative to the amount of data allocated would cause inconsistent and unpredictable pause times in programs that would violate any real-time constraints. By using time-based scheduling, the Metronome garbage collector can achieve systematic, predictable, short pauses of no more than one millisecond to complete its garbage-collection cycle.

Application thread stopping and starting

A garbage-collection cycle consists of a number of increments within which a series of work units are completed. To meet real-time pause requirements by keeping individual garbage-collection quanta times low, each of these work units must be a known measurable quantity of work, so that at each step, the garbage collector can determine whether it should proceed with the next work unit or yield to the mutator until the next scheduled garbage-collection pause to continue. Consequently, an overhead is associated with each garbage-collection quantum to track its time, thereby increasing the overall length of the garbage-collection cycle. There is also the additional overhead of stopping and starting all application threads for each garbage-collection quantum. Because the J9 JVM uses a cooperative suspend model for application threads, which allows the garbage-collection tracing to be accurate, there is an associated overhead with stopping or starting threads. Both of these overheads can affect throughput.

Root scanning

Work units within a garbage-collection quantum can consist of a number of different operations. Generally, each is a known measurable quantity with maximum path lengths whose cost can be evaluated to determine whether the garbage collector should proceed or yield. However, there are some work units to which the cost cannot be easily ascertained, and these cases should be guarded against when writing application code. These problematic cases relate to threads and their corresponding structures. Thread stacks can be complicated and time consuming to scan, and sufficiently deep stacks can be the source of outliers in garbage-collection pause times. Thread-local Java Native Interface (JNI) references, along with the thread stacks, must be scanned as a single atomic unit. If there are a sufficiently large enough number of JNI local references on a thread, the pause times could exceed the targeted value for a quantum.

Allocation

Allocation of objects in the Metronome garbage collector is performed using segregated free lists to manage the available memory.[4] The heap is divided into a series of evenly sized pages that represent a size class from which objects can be allocated. These heap pages are used to create individual units of work so that the Metronome garbage collector can schedule operations on a page with predictable time requirements to complete the operations. The page- and size-class splitting is calculated so that in a worst-case scenario, no more than one-eighth of the heap (12.5 percent) would be lost because of fragmentation or unused ranges of memory due to objects smaller than the size class being allocated. In practice, this number rarely exceeds two percent.[5]

Arraylets

An area of concern in any collector is the handling of large objects, particularly arrays. Although enough total free memory might be available to handle an allocation, there might not be enough contiguous free memory within which to lay the object out, in which case, the garbage collector performs a heap compaction, which can be time consuming and not easily incrementalized. The Metronome garbage collector uses an array-splitting technique called *arraylets* to lay array objects out in memory. Arraylets are hierarchical representations of arrays that enable array memory to be allocated individually (leaves) with a central object representing the entire array (spine). By splitting the array up into separately allocatable chunks, you can take advantage of the heap layout to avoid the need for contiguous storage for large objects, and consequently avoid having to start and complete garbage-collection cycles for the sole purpose of freeing memory to satisfy the allocation.

Write barriers

The Metronome garbage collector is an incremental collector that achieves a full collection by stopping the virtual machine at consistent intervals and performing a small amount of work in each interval. The Metronome garbage collector uses a variant of the Yuasa snapshot-at-the-beginning method,[6] which incurs a level of overhead associated with each object assignment into the heap; as object references between one another are created and destroyed, the virtual machine manages these changes for the garbage collector to reconcile. A nonincremental garbage collector would not incur this management overhead.

Multiple JVM support

The garbage collector is dynamically adjusted to enable multiple real-time JVMs to run on the same system. It runs just above the priority of the highest real-time thread, and as a result, the garbage collector reflects the priority assigned to the real-time threads in that JVM. By assigning priorities appropriately, it is possible to configure a set of threads in one JVM to have priority over another set of threads in a different JVM including the garbage-collection threads, which reflect the priority of the application threads in each JVM. Along with the ability to bind a JVM to a subset of the processors, this dynamic priority assignment facilitates running multiple JVMs in the same system that can be required in more-complex systems.

Real-time compilation

Most JVMs employ a JIT compiler to generate native code for frequently used methods as the application runs for several reasons: to eliminate the overhead of bytecode interpretation, to take advantage of the strengths of the processor's native instruction set and to exploit dynamic application characteristics observed when a particular program runs. Modern JIT compilers use the same technology developed to compile static languages, such as C/C++ or Fortran, as well as new technologies targeted at optimizing the performance of Java programs. These new technologies are often speculative in nature because they must account for the dynamic class-loading support required by the Java language,[7] and they often sacrifice worst-case performance to improve average-case performance. This focus on average-case performance is one reason why traditional Java JIT compilers cannot be used in a real-time environment, where worst-case performance is a critical metric. In addition, traditional JIT compilers run at the same time as the application, randomly consuming resources to compile methods and shattering the predictability required for real-time applications.

WebSphere Real Time provides two forms of native compilation suitable for different classes of real-time applications. The first form is a JIT compiler that has been adapted to avoid speculative optimizations and run at a priority level below real-time tasks. The second form is an AOT compiler that generates Java technology-conformant native code before the program runs.

JIT compilation for real time

The JIT compiler included with WebSphere Real Time performs compilations on a dedicated compilation thread operating below real-time thread priorities. After a method has been identified for compilation, a request to compile the method is placed on a compilation queue and the method continues to run, unlike some commercial JVMs, which would delay running the method until the compilation was complete. Because of its low priority, the compilation thread does not interfere with running the real-time thread. Any real-time thread requiring the processor will immediately preempt the compilation thread.

Many of the speculative optimizations employed by IBM's standard Java JIT compiler are not used, helping to ensure the performance of the generated code runs predictably. In particular, the real-time JIT compiler does not perform aggressive optimizations based on the state of the class hierarchy, which can change as the compiler runs.

AOT compilation for real time

The JIT compilation model works well in softer real-time environments where the presence of the JIT compiler can be tolerated. For harder real-time applications with stringent resource and response-time demands, even a low-priority JIT compiler is not a feasible option but native execution speed is still desirable. Such applications can employ the AOT compiler included in the WebSphere Real Time product. With this compiler, the application's Java bytecodes can be compiled to native code before the program is run and stored into a Java Executable (JXE) format, which is an efficient storage vehicle for AOT-compiled code. When the compiler is running, the native code is loaded into the JVM, and after some processing to bind that code into the currently running JVM, the code can be directly implemented.

Because the Java Language Specification [7] requires dynamic class resolution, AOT-compiled code cannot include any assumptions about field offsets within objects, or the targets of invocations. Therefore, AOT-compiled code is generally slower than JIT-compiled code, because many compiler optimizations rely on precise information about fields and methods. Nonetheless, AOT-compiled code is almost always faster than bytecode interpretation, so in cases where a JIT compiler is not viable, AOT compilation is frequently a desirable alternative. Furthermore, because AOT compilation time is not a runtime cost, more methods can be compiled with an AOT compiler than with a JIT compiler, which can result in an AOT-compiled application running faster than a JIT-compiled application.

Real-time middleware

A key component of any deterministic enterprise solution is the real-time messaging middleware. This software is the backbone by which real-time, critical service providers, service consumers and complex event-processing applications can communicate. These middleware applications have well-defined quality-of-service agreement policies that establish the worst-case total time from start to completion of a message or event. Without these well-defined quality-of-service policies, latency determinism and predictability are unachievable for either a node-to-node or end-to-end real-time, critical business process.

As part of a total real-time enterprise solution, IBM has teamed with key real-time middleware technology providers whose solutions and products are based on open standards from the Object Management Group (OMG). These providers include Real Time Innovations (RTI) with their Data Distribution Service (DDS) technology-based real-time middleware, called *Naval Data Distribution Service* (NDDS, Version 4.1), and PrismTech's OpenFusion RTOrb.

Historically these solutions were targeted towards a C or C++ technology-based programming model, because both were the only languages in wide use that could meet both hard and soft real-time requirements for critical applications. However, with the advancements in real-time Java, these middleware solutions have been extended to support intercommunication between existing C or C++ and real-time Java technology-based applications. A key example of this is the DDG-1000 Destroyer Total Ship Computing Environment (TSCE) in which existing C++ real-time applications need to interoperate with new real-time Java technology-based applications as part of the ship's end-to-end weapons-system integration. To achieve this integration, both RTI and PrismTech have developed interfaces for C++ and Java technology-based applications to communicate. These interfaces map critical thread and method information, such as priority number, method call format and data type from Java to C++. Specifically, for interoperability between a real-time JVM and C++ application, RTI has created a package[8] called *com.rti.ndds.rtsj*, which fully supports the RTSJ for standard Java and RTSJ thread types, as well as RTSJ memory areas. To program with the RTSJ technology-supported DDS middleware, four specific RTI DDS RTSJ classes are provided. RtsjProperty_t provides services to manage all threads created by DDS. RtsjThreadproperty_t provides services to configure threads created by DDS. RtsjThreadSupport provides services to configure domain participants for use with real-time threads and custom memory areas. And ThreadKind is a class that provides an opaque type that provides type-safe enumeration of different types of RTSJ threads.

Figure 2 depicts how C++ and WebSphere Real Time technology-based Java applications can communicate across different languages using the NDDS real-time middleware environment. Here, both C++ and real-time Java components are anchored to different domain containers that publish or subscribe to services. Data can be sent to one or more subscribers by a publisher as long as they have the same topic. The domain container also defines the quality-of-service policy parameters and agreements between a set of nodes, as well as communication between nodes (such as publish, subscribe or both).

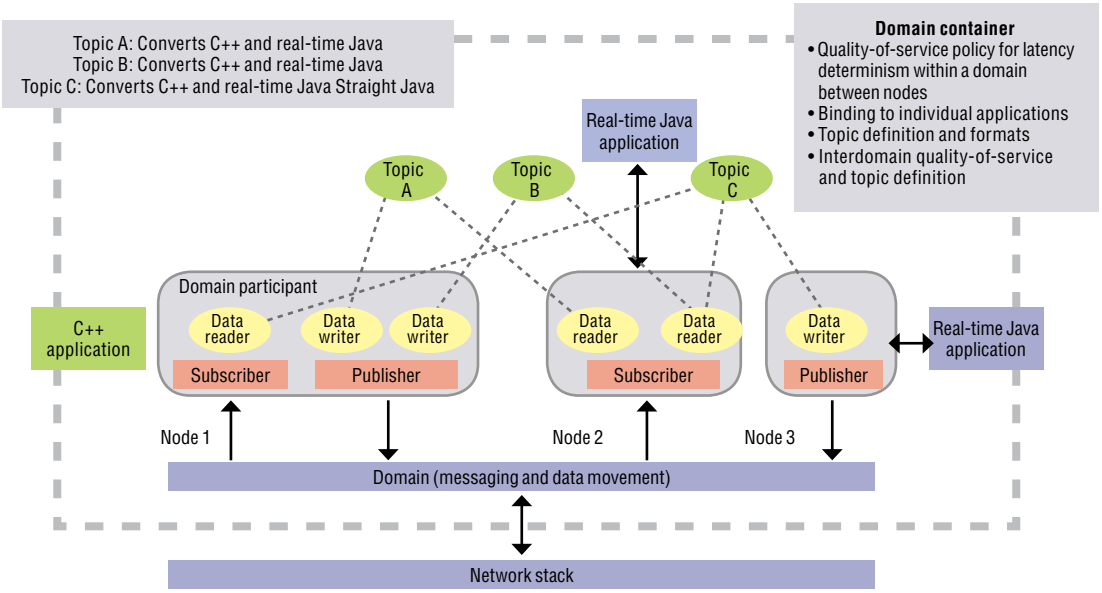

Figure 2. Interoperability between NDDS C++ and real-time Java entities

Practical applications

Many features of WebSphere Real Time are useful to programmers who need to target a traditional operating system. Incremental garbage collection and priority-based threads would clearly be useful in many applications, even if hard real-time guarantees could not be met and only soft real-time performance was available. For example, many would welcome providing an application server that could provide predictable performance without unpredictable garbage-collection delays. Similarly, enabling applications to run high-priority Java health-monitor threads with reasonable scheduling guarantees would make Java server development easier.

The current WebSphere Real Time product provides a full-function runtime environment. It was designed to provide tools to make it easy to eliminate unpredictable delays in applications due to class loading and compilation. Providing tools to trace paths from the operating system through the JVM and into applications makes it easier to perform detailed performance analysis. WebSphere Real Time offers new and innovative ways to develop submillisecond Java critical regions without hand-coded memory management. You can download the first of these technologies from the alphaWorks Web site at www.alphaworks.ibm.com/topics/realtimejava.

Other key technology components are required to create a comprehensive enterprise infrastructure, including security, system management, information management, development, governance and runtime environments.

Summary

This white paper defines soft and hard real-time applications and predictable performance, and presents the features of traditional JVMs that create unpredictable delays while an application runs, including class loading, compilation, garbage collection and thread management. It also discusses how the WebSphere Real Time solution, in conjunction with a Linux distribution containing real-time capabilities, and tools from the IBM alphaWorks Web site, addresses each of these issues. Static precompilation of code helps ensure that no compilation is required at run time.

Alternatively, tooling to generate code that loads and compiles referenced classes at startup is provided. The IBM Metronome garbage collector delivers an innovative solution to the problem by performing very small increments of garbage collection frequently, helping to eliminate large pauses and replacing them with frequent, small pauses. Finally, IBM's support of RTSJ enables programmers to have exacting control of the threads they create and the periods they run at, with precise control over thread priority, preemption and priority inversion.

For more information

To learn more about IBM WebSphere Real-Time software, contact your IBM representative or IBM Business Partner, or visit:

ibm.com/software/webservers/realtime/

To join the Global WebSphere Community, visit:

www.websphere.org

IBM Corporation
Software Group
Route 100
Somers, NY 10589
U.S.A.

Produced in the United States of America
03-07

[1] Gosling, J., B. Joy, G. Steele and G. Bracha. “Java Language Specification, Third Edition.” Addison-Wesley, 2005.

[2] Bollella, G., J. Gosling, B. Brosgol, P. Dibble, S. Furr and M. Turnbull. “The Real-Time Specification for Java.” Addison-Wesley, 2000.

[3] Bacon, D., P. Chengg and V. Rajan. “A Real-Time garbage collector with low overhead and consistent utilization.” Proceedings of the 30th Annual ACM SIGPLANSIGACT Symposium on Principles of Programming Languages. 2003.

[4] Bacon, D., P. Chengg and V. Rajan. “A Real-Time garbage collector with low overhead and consistent utilization.” Proceedings of the 30th Annual ACM SIGPLANSIGACT Symposium on Principles of Programming Languages. 2003.

[5] Bacon, D., P. Chengg and V. Rajan. “A Real-Time garbage collector with low overhead and consistent utilization.” Proceedings of the 30th Annual ACM SIGPLANSIGACT Symposium on Principles of Programming Languages. 2003.

[6] Yuasa, T., “Real-time garbage collection on general-purpose machines.” Journal of Systems and Software 11: 3 (March 1990), 181–198.

[7] Bollella, G., J. Gosling, B. Brosgol, P. Dibble, S. Furr and M. Turnbull. “The Real-Time Specification for Java.” Addison-Wesley, 2000.

[8] RTI DDS RTSJ API Reference Manual. 12 October 2006.

WSW11305-USEN-00