Real-time Operating Systems that Support Java

W. Arnold, K. Branson, D. Chung, R. Pesigan
University of California, San Diego

1. Introduction

Embedded systems and devices are becoming more common as processors become more sophisticated, powerful, and cheap. Items such as cellular phones, personal digital assistants, microwave ovens, and even automobiles contain embedded systems. Such systems interact with the real world and its real-time constraints. In order for a system to be real-time, it must respond to these asynchronous external events and provide service guarantees within an expected amount of time. For example, embedded systems are often implemented using multiple running tasks (or threads) to deal with the random or periodic properties of these external events. As the system becomes more complex, the number of threads needed to manage these events becomes even more complicated. From the programmer's perspective, managing these threads can be burdensome and onerous. Fortunately, tools have been made available to ease programming in this thread-level environment. The most popular software tool available is a real-time operating system (RTOS). Unlike operating systems running on desktop computers or time-sharing systems, real-time operating systems provide the functionality and support for real-time scheduling and thread/task management more efficiently. Often, real-time operating systems are dedicated to one application, can respond to asynchronous events, and support a limited number of extensions for better performance.

However, programming in a RTOS is widely understood to be difficult and very costly. One difficulty arises because many real-time systems are processor dependent, requiring worst-case analysis be repeated for each version of a processor. Also, many real-time systems use proprietary operating systems that are specialized for the system's hardware. This requires the programmer to relearn a new operating system for each real-time system. In addition, high-level tools are not always available for these specialized environments. Instead the analysis, profiling or debugging is done by hand. Many times, trial and error, rather than formal methodologies, is used to get the best worst-case performance. This analysis, debugging, and profiling must be repeated every time the application code is ported to new hardware or a new operating system [7]. Given these difficult requirements, common programming errors only further exacerbate the problem.

The Java language contains characteristics that make it attractive for embedded system programming. Java is a simple, object-oriented programming language. Java is easy to learn because it is has similar constructs to the commonly used C/C++ language. Java eliminates the troublesome C and C++ constructs that cause common programming errors: weak typing and referencing memory through pointers. Java eliminates some runtime errors by providing run-time checks, such as array bounds checking. Java architecture neutral classes provide the portability that popular embedded system language such as C and C++ lack. Java supports multi-threading in the language itself. Because embedded systems usually run using concurrently executing threads, Java has built-in support for multi-threading, providing a standard threads interface [1].

Unfortunately, the current implementation of Java and the JVM specification are not sufficient for real-time systems. There is no task scheduling defined in the traditional Java environment. Java Virtual Machine tasks such as garbage collection, dynamic loading, and compilation

have variable run-times. These tasks can pre-empt Java threads causing the thread to be unreliable in its response times. There are no methods for specifically reserving and utilizing resources [7]. The question arises: given these limitations, can real-time operating systems support Java?

In Section 2 we describe what a real-time operating system is in more detail, describing its characteristics and differences from conventional operating systems. In Section 3, we survey four different operating systems that support Java applications (JN, Joust, real-time Java threads, and Sun's JavaOS), three of which support real-time Java. In Section 4, we analyze each approach and discuss advantages and disadvantages in their architecture. In Section 5 we state our conclusion.

2. Real time Operating Systems

A real-time system must respond to asynchronous, external events with a guarantee of not only the correctness, but also the timeliness, of the response. Depending on the system, a late response can have catastrophic results or just seriously downgrade the system's performance. Systems that *must* respond to an event by a deadline are hard real-time systems. An example of a hard real-time system is one that controls a car engine, as a failure to meet a deadline would cause the engine to fail. Systems that are not hard real-time but are dependent on deadlines being met are soft real-time. An example of a soft real-time system is a "point of sale" system in a store, which is used to scan barcodes. If the system cannot respond in time, an error can be signaled and the barcode can be rescanned [9].

A real-time operating system is an operating system on which a real-time program can be implemented to make the program real-time. Real-time operating systems are designed for a variety of applications with different requirements and programming resources. This means that the choice of operating system is very important. Often, a unique real-time operating system is designed for a specific application. The advantage of this is clear: the requirements of the application are considered during the design of the operating system, so the system can run optimally under the conditions imposed upon it during the running of this application. For example, if the operating system designer knows that his application is a fairly simple system but does not have much memory and will need to support high task concurrency, he will choose to sacrifice some functionality to reduce the overhead for context switching. However, if the operating system designer knows that his system will have more resources and require more complex functionality, he may decide to allow for a relatively high overhead for context switching. There are advantages of using commercial real-time operating systems designed to support many applications as well: there is no need to design an operating system as well as maintain and upgrade it. The commercial operating system designer does all this [3]. In addition, some real-time operating systems were not originally designed to be real-time at all – they are established operating systems that have been changed in some way to meet real-time requirements. An established example of this is a Unix operating system that has been modified to be real-time. This approach facilitates the writing and testing of application programs.

No matter what application it is designed for, real-time operating systems require certain characteristics. These include *interrupt handling*, the ability to *memory-lock* programs, and *multi-tasking* [3], to handle the asynchronous arrival of events, as well as task communication and synchronization, modularity, and information hiding.

First, it is vital that bounded interrupt handling, with varying interrupt priorities, be supported by a real-time operating system [9].

Interrupt handling allows the system to detect an event and respond to it immediately, by interrupting and suspending what it was doing before and catering to the new event. When an event occurs, because it is asynchronous and unpredictable, the system can be in any state. The system must detect the event and correctly respond to it in a certain amount of time. A currently running process might then need to be interrupted so that the resources used by it can be used to service this new event. To be able to determine whether the task corresponding to the new event should preempt the currently running task, a real-time operating system must support interrupts and tasks of varying priorities. In order to bound interrupt handling, that is, to limit the amount of time that the system is given to respond to an event, not only the speed of the computation that must be performed, but also the overhead required interrupting a task and switching to a new one must be considered. The time that it takes for the old task to be suspended and the new task to begin running depends on the *interrupt latency* and the *context switching* time. Because of the asynchronous nature of events, the currently running task could currently be uninterruptible because it is performing an action that must be atomic. The maximum amount of time that could be spent in an uninterruptible portion is the maximum amount of time that the system could have to wait before switching to the new task, and is thus the *interrupt latency* that must be considered. The amount of time that the system then requires to save the state of the current task and then load in the new state of the new task is the *context switching* time [3].

Second, a real-time operating system must be able to identify certain programs as *memory-locked*. A memory-locked program must be kept in main memory and not stored on disk. This prevents context switching from occasionally requiring a disk access and thus being incredibly slow [3].

Third, a real-time operating system must be able to support multi-tasking. That is, it must be able to allow multiple tasks to run concurrently [3]. Multi-tasking is required so that the operating system can respond to asynchronous events immediately without losing the previous work done on other tasks it is performing. It is also necessary to exploit the inherent parallelism present in many tasks. This allows a task to be decomposed into many smaller tasks that can be run in parallel and perhaps faster than if the original task had been run sequentially. This makes it essential that a mechanism for task communication and synchronization be provided by a real-time operating system [2].

Finally, modularity and information hiding is needed to make a system more modifiable, maintainable, and reliable. If a design decision is made visible only to a single module, then it is easier to modify the system as well as maintain it. Modifying a system is especially essential in a commercial system designed for many applications. It allows the application programmer who uses the operating system to add application specific code [2].

3. Operating Systems

3.1 Real Time Java Threads

The following section is from A. Miyoshi's paper [5].

To create a real-time environment that could support Java applications, Miyoshi et al extended the general Java Virtual Machine (JVM) to support *real-time Java threads* and real-time Java thread synchronization, making as few changes as possible to the general JVM and the Java language itself.

These extensions to Java include a new class, RtThread, which extends the Java Thread class. The programming interface to create and run real-time Java threads is similar to the interface for regular Java threads, except for added parameters that allow the

programmer to declare timing constraints. Real-time Java thread creation requires extra parameters in both the constructor and the start() method. The constructor requires parameters to indicate the priority, start-time, period, and deadline of the thread. All non real-time threads have the lowest priorities, followed by house keeping activities like garbage collection, and finally followed by real-time threads. This allows real-time threads to run without interruption from many requirements of Java as well as activities that do not have specified time constraints The start() method requires parameters specifying the deadline handler and the function to be called if the deadline is missed. In addition, other member functions are provided for the RtThread class that allows the programmer access to these real-time features of a thread. To support the RtThread class, two additional classes were added. The first, the RtHandler class, is required to specify handlers to detect deadline misses of real-time threads and handle them as specified by the programmer. A deadline handler is another real-time thread. If a deadline is missed by a real-time thread, that thread is suspended and the specified deadline handler runs. The deadline handler can perform recovery actions and has access to the original thread. The second, the Time class, is required to allow timing constraints to be specified.

This real-time Java environment is accomplished by implementing a Java server, which supports Java applications, as a user-level server on a real-time kernel, the RT-Mach microkernel. RT-Mach extends the Mach 3.0 microkernel to provide real-time features. RT-Mach provides mechanisms to support real-time threads, synchronization, scheduling, IPC, and resource management. Many of these features are provided by the Real-Time Server, another user-level server on RT-Mach, on which the previously mentioned Java server is based. The Java server contains the mechanisms of the Real-Time Server as well as a mechanism to interpret Java bytecodes.

The real-time Java thread implementation, described above, is based on the RT-Threads, the kernel-level real-time threads provided by RT-Mach, which also require specification of the priority, start-time, period, and deadline of the thread. When the start() method of a member of the RtThread class is called, an RT-Thread with the timing specifications of the Java thread is created. This allows Java threads to be scheduled using the real-time scheduler already implemented in RT-Mach.

To accomplish real-time synchronization, the rt_mutex_lock and rt_mutex_unlock primitives provided by RT-Mach are used. As Java requires nested locking and the RT-Mach primitives do not, extensions were required of the JVM to keep track of the current lock owner and how many times the lock has been locked. Unfortunately, deadlocks can also arise when a real-time thread inside a Java synchronized method misses its deadline and control is given to the deadline handler. The deadlock occurs if the thread has not released its lock and the deadline handler tries to acquire it. To avoid this problem, the thread continues to execute even if it misses its deadline, until it releases the lock. At that point, the deadline-handler takes over and can acquire the lock without any problems.

This implementation of real-time Java threads was found to meet specified timing requirements, but significantly slower than RT-Mach's implementation of real-time threads. This lag is due to the JVM support of nested locking, garbage collection, overall system overhead, and the use of kernel-level threads and real-time locks instead of user-level threads and locks. Work continues to be done to add support for networking and real-time garbage collection.

3.2 JN Operating System

The following section is from B. Montague's paper [6].

The Java Nanokernel (JN) is an operating system for an embedded Java network computer. It is designed to support the Java Virtual Machine on a small "single-chip" embedded in a PC that is attached to the Internet.

The design of JN is based on that of a soft real-time kernel used on earlier systems such as the kernel of IBM's TSS OS. The JN implementation is structured into layers, as seen in the figure below. The lowest layer contains the hardware device drivers. The second layer implements the JN nanokernel. The third layer implements the APIs that provide the low-level used by the JVM and KA9Q TCP/IP stack, which supports networking. The final layer represents the applications: the JVM, the KA9Q TCP/IP Server threads, and Java Threads. Implementing the low-level API's as a layer of the JN allows the JVM to run as an application.

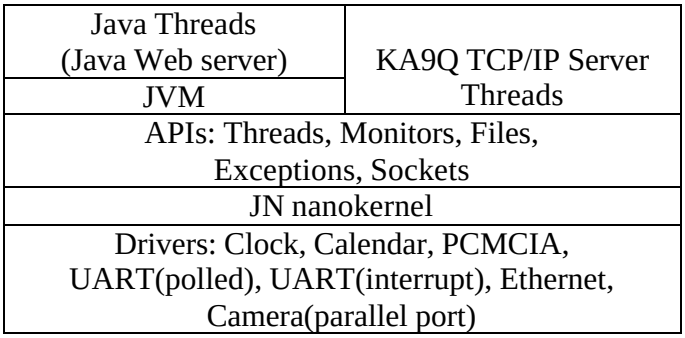

Figure: Java Nanokernel and Components [6]

The second layer of JN, the JN nanokernel, is optimized for the support of the JVM and Java. As a full kernel was not needed to support the JVM, only a nanokernel was implemented. A nanokernel is unique from a full kernel or even a microkernel in that it has an extremely limited functionality as it implements only one required API (a fork routine), does not provide a generic name-space or I/O support, does not specify a message-based IPC mechanism, and supports only light-weight threads without memory protection, process specific page tables, and any concept of resources private to a thread. Thus, the JN nanokernel was very small and its implementation required under 2000 lines of C code. The JN file system is flat and also very simple, but provides the hierarchical directory system necessary for the hierarchical class structure of Java.

The third layer, the APIs, are implemented to provide services to run the JVM as an application and to provide services necessary to support the KA9Q TCP/IP stack. There are four major classes of APIs that support threads, monitors, files, and exceptions. Thread API routines are similar to Unix thread API's, except that they allow the JN scheduler to be disabled and enabled. A thread about to perform time-critical actions can disable the scheduler to preclude other threads from running and using resources, then re-enable the scheduler when those time-critical actions are finished. Monitor APIs are conventional as well and are used for thread synchronization. File APIs include many Unix file operations, such as open, close, read, write, and lseek. JN also provides the available() call which can be used to determine the amount of space left in a file. Exception APIs provide some of the functionality of Unix signals. Interrupt handlers can be specified to be associated with signals, as in Unix. Threads can disable and

enable interrupts. The implementation of this requires that each thread have a non-interrupt context and a software-interrupt context. If a software interrupt is queued while a thread is running normally, the non-interrupt context of the thread is saved and the soft-interrupt context of the thread is loaded. The interrupt handler then runs to completion, at which time the non-interrupt context of the thread is reloaded and the interrupted thread continues to run.

The structuring of JVM in JN takes the "reverse engineering approach." This means that all the functionality of Green Threads, a multi-threading C runtime, has not been fully implemented. Instead, some of the required functionality for Green Threads is added during JVM implementation phase. It is worth noting that the JN JVM has been modified to support real-time applications. For example, running the standard garbage collection mechanism while also supporting real-time applications could lead to the garbage collection preventing real-time guarantees from being met. For instance, In Embedded-Java Web-Camera worked on by the UCSC embedded system research group, the garbage collection mechanism could cause TCP/IP threads to be blocked for long periods while garbage collection is running. To remedy this inappropriate situation, two approaches are considered.

The first approach is to extend priority levels. Java currently uses 10 priority levels. Threads with larger priority run before those with smaller priority. JVM originally gives TCP/IP threads a high priority within this range. However, the JVM garbage collector disables thread scheduling when it runs, to assure unrestricted access to the heap. TCP/IP threads should not be disabled for a long period time because of garbage collector. The current JVM in the system added another 6 priority levels to run TCP/IP threads at a level higher than any Java thread.

The second approach to allow garbage collection to be performed while real-time applications run is to modify the garbage collection mechanism. The standard garbage collector makes multiple passes over the heap, but the modified garbage collector makes a single pass. Although this new single pass may take longer than the original `lazy' method of garbage collection, it collects space more efficiently and, in the long run, results in fewer calls to the collector.

3.3 Joust

The following section is from J. Hartman's paper [4].

Joust is a real-time environment optimized for the support of liquid software written in the Java programming language. *Liquid software* is portable code designed for low-level, communication-oriented system. In liquid software, it is more important that data packing, transfer, and unpacking be fast than that computation be fast. Thus, the demands of liquid software on a system are very different from the demands of most applications. First, liquid software requires the ability to specify and meet timing deadlines, as these are inherent in communications applications. In order to meet timing deadlines, the system on which liquid software runs must provide it with the ability to directly manipulate resource allocation. For the allocation of CPU time, I/O buffers, and link bandwidth to be adapted for the software being run, the system's allocation policies must be very flexible. In contrast, most general-purpose operating systems, as well as high-level languages like Java, try to hide their resource allocation methods. Second, minimizing the cost of abstraction interface overheads is particularly important for maximizing the performance of liquid software. Since liquid software is primarily oriented toward data moving, packing, and unpacking, system services are

highly used and high overheads can greatly downgrade system performance.

Joust provides high performance for liquid software written in Java by combining a general-purpose operating system and a runtime system. To do this, it combines the Scout OS, the Java Virtual Machine (JVM), and the Just-in-Time (JIT) compiler into one operating system.

The Scout OS kernel consists of independent, low-level communication modules. For example, a single module might implement a single networking protocol. Scout can be configured to support many different types of applications because these modules can be connected into a *module graph*, whose nodes are individual modules and whose edges are the dependencies between the modules it connects. Because the modules are independent and well defined, the only limitation on which modules can be connected is that the output of one module be of the same type as the input of the other, that is, they share the same *service interface*. Different module graphs can be configured to support many different purposes, such as active network nodes, multimedia displays, and web and file servers.

Scout allows *paths* through module graphs to be defined. These paths describe the way data flows through the modules. The first module of a path is applied to the input data, then the second module of a path is applied to the output of the first module, then the third module of a path is applied to the output of the second module, and so on. Paths are created dynamically when I/O connections are opened. A set of attributes and a starting module correspond to the specific connection and the type of data being transferred. These attributes and starting module are used to determine the path incrementally: starting with the first module, each module uses the attributes to deterministically determine which module adjacent to it in the module graph to be next. When data is received from an I/O connection, the data is associated with a specific path. Scout schedules a thread to move the data along this path. This thread sequentially runs each module of the path on the data. Scout limits the memory-usage of a path by limiting the sizes of its input and output queues. Scout limits the CPU time of a particular path by choosing when the path executes. Each path has an attribute defining whether the threads that execute on it are real-time, in which case they have a specified deadline, or best effort, in which case they are to run as fast as possible. The Scout scheduler schedules both real-time and non real-time threads by letting each thread run for a fraction of the total CPU time. However, real-time threads meet their specified deadlines because the Scout scheduler allows real-time threads to steal cycles from lower priority threads.

The Joust system has a static module graph that is optimized for liquid software. It also contains a module that implements the JVM, tailored for Scout as well as its liquid software applications. Scout must provide quality of service and real-time deadlines. Given that the JVM is a module of Scout, the JVM must have high enough performance so that it does not slow down the entire system. The Java Virtual Machine does not deliver quality of service guarantees or real-time deadlines, but its performance cannot be an impediment to Scout providing these services. Thus some modifications needed to be made to both Scout and the JVM.

Modifications to the JVM included optimizations made to slower functions that would be highly used by liquid software, such as the synchronization and exception handling functions. These modifications were only to the implementation of the JVM, not its functionality. For example, Java allows nested locking, via monitors and condition variables. Since monitors are usually unnecessary, given that most Java applications are single-threaded, the Scout JVM optimizes

the locking implementation for this case. The Scout JVM does this by keeping a reference count of the number of times a monitor has been nested, rather than nested locks. The Joust JVM also has a different implementation of exceptions. The Joust JVM allows a thread to exit a critical section before catching an exception from any thread. If an exception was caught before completion of a critical section, control could jump over the critical section and the critical section might never be executed, potentially causing deadlock.

The JVM was also optimized for use in the Scout path model. The Joust JVM has mapped the entire Java API to the Scout OS API, with the exception of the System.exec command. The exec command allows programs to execute commands specific to a platform and is incompatible with Scout. Some of the functionality of the Java API was different from that of the Scout API. These differences needed to be accommodated for in the mapping from Java APIs to Scout APIs. For example, sockets in a JVM deliver data synchronously, meaning the socket blocks until data arrives. Scout's sockets deliver data asynchronously. The Scout module for sockets is modified to buffer the data until the receiving module is ready to receive the data. Another difference between JVM and Scout is that the JVM supports pre-emptive threads, while Scout does not. The Joust JVM compiler was modified to insert thread yields on backward edges on the control flow graph. (**?)** Currently, only a single Java application can be run in the Joust JVM, as a security policy for multiple, untrusted Java applications has not yet been devised.

In addition, the JVM has been extended to add two APIs to implement Scout's createPath and extendPath functions that create and modify paths between Scout modules.

However, even with the improvements to the JVM and the modifications to Scout, a limitation of Joust is that the JVM module cannot be used in a real-time path, given that the JVM cannot guarantee any real-time service. However, this does not limit Java applications that can run on Joust. For instance, to build a MPEG viewer in Java, the Java program first builds the path in Scout to handle transporting the MPEG frames from the source (network, file) to the display device. It is the modules in Scout's path that ensure that the delivery of the frames from the source to the destination will be real-time. The rest of the Java application handles retrieving the frames from the display device and displaying them in a window.

Joust imported the Toba translation WAT (way-ahead-of-time) compiler and also added a JIT (just-in-time) compiler to increase the JVM performance. WAT compilers assume that compiling the classes ahead of time and storing the result can accelerate the execution of a large part of frequently executed Java classes (such as java.lang or java.awt). However, WAT compilers do not support dynamic linking and loading of classes at run-time, including user programs. These classes cannot use the WAT compiler and must use the JIT compiler instead. Both the WAT and JIT compilers improve speed of a Java program by more than one magnitude. However, the Joust JIT compiler is not as fast as other JIT compilers. This is because the Joust JIT compiler was written in Java, while standard JIT compilers are written in C. The JIT performs faster when written in Java because JIT compilers use bit manipulation, which is slower in Java than in C.

3.4 JavaOS

The following section is from Stuart Ritchie's paper [8].

JavaOS is a stripped down operating system that runs Java programs directly on the hardware. Implementing Java directly on the machine allows Java to run without a host operating system, which allows the overall system to be smaller and faster since often full

operating systems implement many features not needed for Java applications. JavaOS was implemented in 4 MB of ROM and 4 MB of RAM. Since JavaOS was originally designed for embedded systems and diskless machines with small memories, this compactness and speed is very important.

JavaOS is structured as a sequence of layers. Implemented directly on top of the hardware is the lowest layer of JavaOS, the Java booter. It is the hardware abstraction layer and contains code to support traps, interrupts, context switching, booting, exceptions, and memory management. The second layer is the Java microkernel, which implements a time-slicing thread scheduler that allows preemption. The third layer is the Java Virtual Machine, which interprets Java bytecodes in a similar manner to the JVM provided with the Java Developers Kit. These three layers are implemented in C. However, only the two lowest layers, the hardware abstraction layer and the microkernel, are platform dependent, as the other layer use only the functionality provided by the layers below them. The rest of JavaOS is written in Java itself, and thus is also platform independent. These include the JavaOS Graphics System, device drivers, the network protocol suite, the JavaOS Windows System, the standard Java classes, and the Java API. JavaOS is small and efficient because the minimalist approach of designing only the functionality required to support the standard Java classes was taken. So much of JavaOS was written in Java to make the system as portable as possible.

This extreme amount of portability of the JavaOS is one of its main advantages. Since only the two lowest layers of JavaOS, the JavaOS booter and the microkernel, are platform dependent, only a small portion of the JavaOS code must be rewritten for each new platform the operating system is to be implemented on. In addition, because much of JavaOS was written in Java, the features of Java were taken advantage of during the development of JavaOS, allowing system development to be quick and painless. The use of Java allowed, for example, the Ethernet device driver to be developed in a single week. System level development was made much easier through the use of Java's object-oriented model, which allowed code to be clean, compact, and organized. All code in Java is in the form of classes, which are structures that group together variables and functions on those variables. Classes in Java can be defined to inherit from other classes. If, for example, there was a Java class representing all network drivers and a class representing the specific Ethernet network driver that was derived from the general class, then methods in the general network driver class could be redefined in the derived Ethernet network driver class, for example the readPacket() and sendPacket() member functions. Code outside of these classes could call the readPacket() and sendPacket() functions without knowing the details of their implementation. Thus, Java allows implementation details to be hidden. This allows increased flexibility. Java also has a good model of event handling which allows an event to be handled by the most specific code to that event. Java also provides automatic garbage collection and memory allocation, which makes management of objects and data structures easy and less error-prone. Furthermore, JavaOS allows the dynamic loading of classes, which allows the operating system to dynamically be extended. Also, there is a plethora of debugging and development tools for Java that simplify implementing the system. Moving much of the implementation of the operating system up into higher layers allowed JavaOS to be more portable and easier to implement than most operating systems, while still providing good performance. The Java implementations of a networking application and a window management application were found to

benchmark at least as fast as their conventional implementations.

While Java and its class libraries provide most of the required API, a few more primitives were required for systems programming. These were primitives for the mutual exclusion lock and the condition variable. While the keyword *synchronized* in Java allows mutual exclusion, it does not provide all the flexibility that can be provided by mutexes and condition variables. Thus, JavaOS provides a Mutex class. Java also provides wait/notify primitives for cooperation between threads, but the flexibility of multiple cooperation points for a single lock is often needed. Thus, JavaOS provides a class to implement condition variables. In addition, to implement device drivers, JavaOS required primitives to allow interrupt handling and load/store operations. While Java does provide event handling, this is only through static, predefined interfaces, like the mouse and keyboard of a computer. Java does not allow software to access the hardware interrupt vector, so JavaOS needed to provide the Interrupt class that allows software to register and deregister handlers. A Memory class was required because Java does not allow software access to memory-mapped device registers, thus JavaOS provides a primitive for this access.

These are the only additions to the Java language and class libraries that were required to implement the JavaOS. The JavaOS eliminates the overhead of a host operating system and thus is smaller and more efficient, was written mostly in Java, is very portable and easy to maintain, and provides a complete environment to support application programs written in Java.

4. Analysis

We have surveyed four different operating systems that support Java applications, three of which were real-time (the real-time Java threads environment, JN, and Joust) and one which was not (JavaOS).

All three real-time operating systems, the real-time Java threads environment, JN, and Joust used a real-time kernel to support the JVM, which runs real-time Java applications. The real-time Java threads environment extends the JVM to include a new real-time thread class, which map to the RT-Mach kernel threads. In contrast, JN implements APIs on top of its real-time nanokernel, which are used by the JVM implementation to support the Java runtime environment. Joust differs from both of these systems in that the real-time work is done by Scout OS and the Java application only deals with the data after the real-time task is completed. In these three cases, Java is made real-time mostly by relying on the real-time features of the kernel. JavaOS, however, is not a real-time operating system. It implements the JVM directly on a hardware platform, without the use of a host operating system like the other three systems. We feel that if this platform supplied real-time functionality, JavaOS could also be made real-time.

We believe the real-time Java threads environment includes all of the characteristics required of a real-time operating system, because it uses a known real-time RT-Mach kernel. A Real-Time Server runs on RT-Mach and provides task management, file management, and interrupt handling. As the Java server in the real-time Java threads environment includes all the functionality of this Real-Time Server, the Java server must also provide task management, file management, and interrupt handling. We assume that since RT-Mach is specified to be real-time, it must include the characteristics required of real-time operating systems: prioritized and bounded interrupt-handling, the ability to memory-lock programs, multi-tasking, and the ability to handle the asynchronous arrival of events. The paper we

read on the real-time Java threads environment specified that task communication and synchronization were provided through the use of functions provided by RT-Mach. The modularity and information hiding required to make an operating system real-time are inherent in the Java programming language and the Java runtime implementation. However, only the ability to specify timing requirements in Java programs is specified in this paper. Thus, we cannot be sure the required interrupt handling is available to the Java applications as well. It is feasible that, if in fact the system hardware does include a disk, program memory-locking be available, since this does not require specification by an application and can be done automatically.

However, we feel that JN *does not* meet the requirements of a real-time operating system. Real-time operating systems require that asynchronous events be serviced within the interrupt bound, no matter what state the system is in. However, JN allows the scheduler to be disabled, particularly during garbage collection. If an event occurs while JN has disabled the scheduler to perform garbage collection, an event could be missed and vital information could be lost, assuming there is no limit on the amount of time the garbage collector runs.

Joust, on the other hand, makes more modest claims about its real-time features and implements all of them, again assuming that the real-time kernel that Joust uses, Scout, meets the requirements of a real-time operating system. Joust only guarantees that the tasks implemented by Scout are real-time, not the Java applications themselves. This is sufficient for Joust's intended applications, liquid software.

We also feel that JavaOS could potentially be made real-time as well. Interrupt handling, memory locking, and real-time synchronization, and the ability to handle the asynchronous arrival of events, if provided by the hardware platform, could all be used by the Java applications. The other requirements, multi-tasking, modularity, and information hiding, are all provided by the Java language.

These four operating systems each have their own advantages. The Java real-time threads environment supports general, real-time Java applications. The real-time Java threads environment was implemented with very minimal changes to the Java runtime environment, as it only adds a few new classes to support real-time applications. Thus, the original core design of the JVM and the Java language is not compromised. In addition, the implementation of the real-time Java threads was exceedingly simple because just maps real-time Java threads to real-time threads of an existing real-time kernel. It also allows the specification of clean-up to be done and routines to be called if deadlines are missed.

JN also is designed to support general Java applications. It also has the advantage that the JVM is tightly integrated into the operating system, as JN only provides the API's necessary for the implementation of the JVM. This differs from the real-time threads environment, which implements all of the functionality of a complete kernel, causing more overhead. As JN has less overhead provided by its host operating system, performance of JN can be better. In addition, JN is very small, allowing it to be integrated into embedded systems with limited resources. This differs from the real-time Java thread environment, again because the host operating system of JN is much smaller than that of the real-time Java threads environment.

Joust is designed specifically for liquid software, so is particularly efficient at supporting these. Because of the functionality provided by Scout in support of communication applications, it is trivial to write a Java application for real-time communication, given that all the real-time work is done in the operating system, rather than the application. In addition, extensions

made to the JVM to create paths through Scout's modules are minimal. In fact, real-time applications written in almost any language can be supported using Scout as long as the language's API could be mapped to Scout's API and the language's API could be extended to handle creating and extending Scout's paths.

While JavaOS is not intended for real-time applications as these other three systems are, it also many properties that make a real-time system based on JavaOS attractive. First, JavaOS is extremely portable, as only a small amount of the JavaOS code is platform dependent. This is in contrast to all three of the other systems, which are complete hardware dependent. Second, JavaOS is very compact, in part because it did not require a host operating system like the other three systems. While JN allowed increased performance because its JVM was tightly integrated into the OS, JavaOS ties the JVM even closer to the hardware. This provides increased performance as well because the system can be more efficient when the overhead of a host operating system is eliminated. In addition, few changes were needed to the Java language and class libraries to implement JavaOS. We suspect that, if JavaOS were made real-time, the only changes necessary to the language would be those required in the other systems. Finally, since much of the code of JavaOS was written in Java itself, the many benefits of Java can be taken advantage of to accelerate and simplify system development and modification.

These systems also have distinct disadvantages. The real-time Java threads environment has the disadvantage that performance of the real-time threads is slow because real-time kernel-level threads are used to support their design rather than user-level threads, even though the RT-Mach supports both. It takes a much longer time creating new real-time Java threads or invoking the real-time synchronization mechanisms (such as locking and unlocking) if kernel-level threads are used. This is justified by comparing performance of the Java server running on RT-Mach versus one that is using the original Java Virtual Machine. Thus, while the real-time Java threads meet their timing requirements, the overhead required to create and run threads might prevent this in the future when requirements are more strict. In addition, real-time Java threads are not portable to any other real-time operating system unless it is running a RT-Mach kernel. This implies that a real-time Java application is not portable as well since the real-time APIs used are dependent on the RT-Mach (cannot really port a real-time Java code to another real-time operating system). In short, the JVM and the RT-Mach are tightly integrated.

While the paper we read on JN was very positive and did not admit to many disadvantages of JN, that we do not feel the JN system can support real-time applications is a crippling disadvantage. This is because the paper claims JN is a real-time environment, designed for real-time applications. We feel that JN cannot support the applications that it was designed to support.

There are also disadvantages to Joust. Given that the JVM cannot be included in a real-time module, Joust is only able to support real-time Java programs that use Scout's predefined communication modules. Therefore, Joust is not a general-purpose solution for an operating system to support Java. Also, any Java program written for Joust is no longer architecture neutral because of the Scout extensions to the JVM.

The main disadvantage of JavaOS is that it is not a real-time operating system. Unlike JN, we believe that JavaOS could be made real-time without too much work.

5. Conclusion

Java as a programming language has many advantages and these benefits could

help ease the development of real-time systems. It has been shown that Java can be supported in real-time propetiary operating systems, however no general portable solution is available. Until Sun standardizes real-time properties for Java it will be difficult for one method to become widely accepted.

[1] E. Bertolissi, C. Preece, *Java in Real-time Applications*, Proceedings of the 10th IEEE Real Time Conference, Beaune, France, 1997.

[2] H. Gomaa, *A Software Design Method for Real-Time Systems*, Communications of the ACM, Volume 27, September 1984.

[3] H. Hindin, W. B. Rauch-Hindin, *Real-time Systems*, Electronic Design, Jan 1983.

[4] J. Hartman, L. Peterson, A. Bavier, P. Bigot, P. Bridges, B. Montz, R. Piltz, T. Proesbsting, O. Spatscheck, *Experiences building a communication-oriented JavaOS*, Software—Practice and Experience, vol 30, 2000.

[5] A. Miyoshi, T. Kitayama, H. Tokuda, *Implementation and Evaluation of Real-Time Java Threads*, IEEE, 1997.

[6] B. Montague, *JN: OS for an Embedded Java Network Computer*, IEEE Micro, 1997.

[7] K. Nilsen, *Adding Real-time Capabilities to Java*, Communications of ACM, June 1998/Vol. 41 No. 6.

[8] S. Ritchie, *Systems Programming in Java*, IEEE Micro, 1997.

[9] D. Stepner, N. Rajan, D. Hui, *Embedded Application Design using a Real-Time OS*, ACM DAC, New Orleans, Louisiana, 1999.