A Real-Time Java Server for Real-Time Mach

Akihiko Miyoshi†
†Keio Research Institute at SFC
Keio University
5322 Endoh Fujisawa Kanagawa, Japan
miyos@sfc.keio.ac.jp

Hideyuki Tokuda†‡
‡Faculty of Environmental Information
Keio University
5322 Endoh Fujisawa Kanagawa, Japan
hxt@sfc.keio.ac.jp

Abstract

We have developed a real-time Java server on the Real-Time Mach microkernel which is suitable for embedded systems and distributed real-time systems. By implementing it as a user-level server on Real-Time Mach, applications such as WWW browsers and embedded applications can both execute Java byte codes. In this paper, we describe the real-time issues in Java and the architecture of our Java server. The real-time extension of the Java server and Java threads using kernel-level and user-level real-time threads was also evaluated.

1 Introduction

Java [1] is an object oriented programming language very similar to C++ developed by Sun Microsystems. It was designed to be used in world-wide distributed computing environments, thus having security features and its code is compiled into architecture neutral byte codes.

To execute Java code on a target machine or device, the virtual machine must interpret or dynamically translate Java byte code in to the target machine code. Java also supports a dynamic loading of classes across a network. Because of these unpredictable behavior during the code execution phase, it is often considered that Java is not appropriate for developing hard and soft real-time applications.

However, Java can provide portable multi-threaded programming interface and window system interface. Java can be used to develop object-oriented soft real-time systems with appropriate support. In this paper, we first discuss basic issues in the Java language for developing real-time applications. We then describe the Java server architecture and its extension based on the kernel level and user-level real-time threads and evaluate its performance.

2 Real-Time Issues in Java

Many issues in using Java for developing real-time applications can be classified into two categories: language specification and its execution environment.

2.1 Language Specification

One of the missing real-time features of the Java language is the ability to specify explicit timing constraints. Since programmers cannot assume the performance of the target device which executes the application or the availability of system resources, specifying explicit timing constraints is very important.

For instance, Java provides `sleep(t)` method where a running thread can suspend at least `t` milliseconds. This method alone is insufficient for many real-time programs. Programmers would rather benefit from `sleep_until(time_of_day)` method, or `within (t) do s except q` construct[2]. Furthermore, explicit use of deadlines in thread attributes and the timing fault handler can be a better extension. By using the timing fault handler, the programmer can get feedback whether it has missed the deadline or not.

Another missing feature of the language specification is the ability to provide resource abstraction. In current Java language it is difficult to express resources such as CPU time, memory, network and I/O bandwidth. Without the notion of resources, it is hard for programmers to maintain a certain level of quality of service (QoS) when a system is in a overload condition, or reserve resources needed for real-time activities.

2.2 Execution Environment

As for the execution environment, there are many unpredictable factors such as dynamic loading, linking, verification of code, garbage collection, and scheduling policies for threads.

Another unpredictable nature is caused by the implementation of Java virtual machine itself. There are two models of implementation of Java virtual machine. One model is to implement it as an application or embed the virtual machine in applications such as WWW browsers. This model relys on the host OS it is on. Another model is to implement it without an host OS. Sun Microsystem's JavaOS[3] is an example of this model. Java programs running on different virtual machine architecture shows different characteristics even on the same hardware. This sometimes is caused by the effect of having a host OS and resulting in crossing of multiple layers of services, or the characteristics of the host OS itself. By using our real-time Java server architecture, we should eliminate these unpredictable effects.

3 Java Server Architecture

Considering the real-time issues in Java which we have discussed in the previous section, it is possible to provide soft real-time environments for Java.

In this section, we first describe the structure of the user-level Java server and then discuss the two execution environment for Java server and its real-time features.

3.1 User-Level Java Server

Our Java virtual machine is implemented as a user-level server on RT-Mach microkernel [4]. RT-Mach is an extension of Mach 3.0 microkernel developed at Carnegie Mellon University. It provides distributed real-time computing environment for a wide range of target machines such as Pentium, SPARC, MIPS, PA-RISC, Power PC architectures. RT-Mach provides real-time features such as real-time threads, real-time scheduler, real-time synchronization, and real-time IPC primitives[5, 6].

By implementing the virtual machine as a server on RT-Mach, it can be used as an engine for running basic operating software or application for embedded systems such as Network Computers. It can also be used from user application such as WWW browsers on another server like the UNIX server. With this architecture, Java programmers can expect same characteristics from their programs whether it is executed in embedded system environment or from an application on a UNIX environment because Java server and UNIX server can coexist on the same machine.

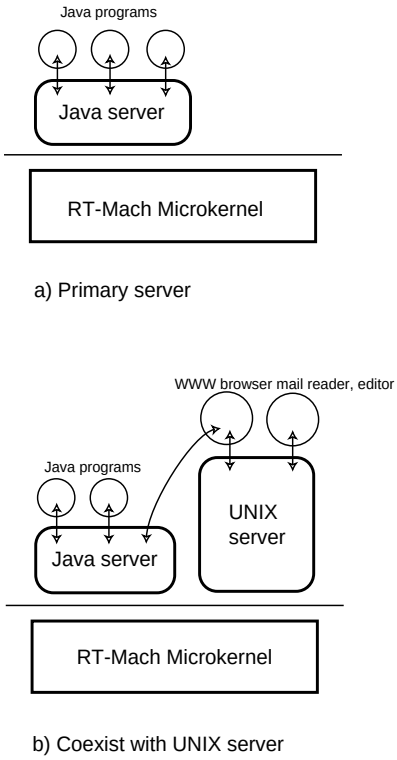


Figure 1. Two Execution Environment for Java Server

Java server is based on RTS (Real-Time Server) [7] which is also a user-level server on RT-Mach. RTS is a simple object-based server which provides task management, file management, name service and exception handling. Java server extends RTS and enables the server to interpret Java byte code as well as native binaries. It has an in-memory file system and different types of file system can be mounted from various media such as a floppy, hard disk or RAM disk. Files can be cached as continuous memory blocks. In our current implementation, Java server will mount Unix file system and copy necessary files (classes) from the local disks into its in-memory file system at initialization. This avoids blocking for disk I/O while executing Java methods, but devices with less memory can choose to read from a mounted file system rather than to keep it in memory or can request data from another server in the same machine or a remote machine across a network. For the Java byte code interpreter, we used a virtual machine called Kaffe[8] which supports inter-

preting as well as JIT.

3.2 Configuration for Java Server

Java server can be booted on RT-Mach as a primary server which is shown in Figure 1a. This configuration will be small enough for devices with limited resources such as Network Computers, personal digital assistants, internet-ready appliances. This configuration is useful for embedded applications and embedded systems written in Java.

Another configuration that can be made is to coexist on RT-Mach with other servers like the UNIX server [9]. It is shown in Figure 1b. Programmers can benefit from this environment because they can use developing and testing tools while running Java applications. Or applications can communicate with the Java server and let it execute Java programs for them instead of embedding the virtual machine in themselves like the WWW browser.

4 Real-Time Threads for Java

4.1 Real-Time Threads

We have extended Java threads for real-time. Real-time threads have been very effective in preserving timing constraints when dealing with continuous media data and thus resulting in decrease of jitters and noise in multimedia applications. Programmers can easily detect whether a thread missed a deadline or not and change the QoS (quality of service) of continuous media dynamically[10].

For example, a movie player application can adjust its performance by choosing to degrade the resolution of the picture or reduce its frame rate when its working threads miss the deadlines specified because of reasons like low performance of the hardware or network, or because of competing jobs in the same machine.

RT-Mach has two implementation of real-time threads. One is kernel-level threads called RT-Threads and the other implemented at the user-level, called RTC-Thread [11, 12]. RT-Thread is scheduled by the kernel-level scheduler and timing management is done at the kernel using a clock device which interrupts the kernel at short intervals. RTC-Thread separated timing management and thread management and put both functions at the user-level for flexibility and efficiency. In both thread models, a thread becomes a real-time thread by specifying its timing attributes.

A kernel-level real-time thread in RT-Mach can be created and killed using the `rt_thread_create` and `thread_terminate` system calls. Unlike non-real-time

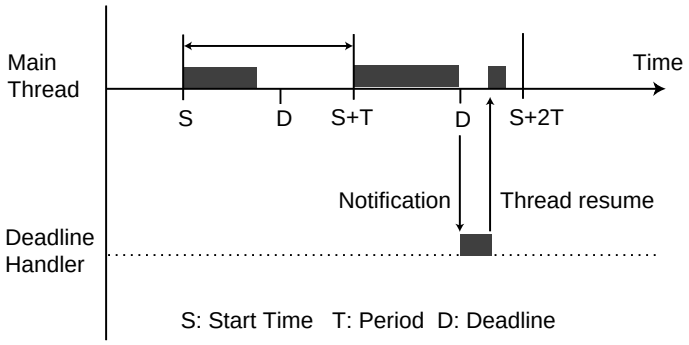


Figure 2. Real-time Thread model

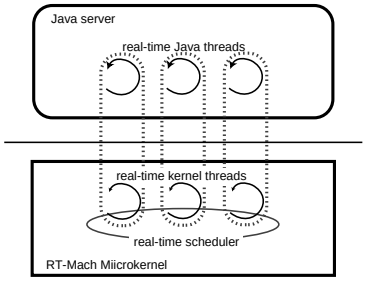
threads, a real-time thread is defined with its timing constraint.

As we shown in a C-like pseudo language in the following example, a real-time thread, `f( )` is created with its thread attributes `f, Si, Ti, Di`. f indicates its thread's function f( ), Si, Ti, and Di indicate thread f's start time, period, and deadline respectively.

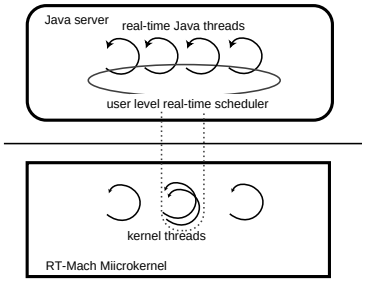
```
1.   root( )
2.  {
3.   thread_id f_id;
4.
5.    f_attr = {f, Si, Ti, Di};
6.         /* set thread attribute of f */
7.    rt_thread_create(&f_id, f_attr);
8.         /* creating f( ) as a thread */
9.  }
10.
11.   f(arg) {
12.    f's body
13. }
```

Note that if thread `f` is periodic then it will automatically restart, or *reincarnate*, when it reaches the end of its function body.

We are implementing two models of real-time Java threads using RT-Threads and RTC-Thread as a base. We inherited the model of real-time threads in RT-Mach shown in Figure 2. Main thread will start at time `S` with a period of time `T`. If it misses its deadline `D` and a deadline handler is specified, main thread will be suspended and thread of control will be handed off to the deadline handler which is another thread. Deadline handler can specify a forward or backward recovery action as well as change the main threads attributes such as priorities, deadline time and period or some application specific attributes. For example, if a thread in a movie player application misses a deadline, the



a) Using real-time kernel threads



b) Using user level scheduler

Figure 3. Kernel-level and User-level Threads in Java Server

deadline handler can choose to reduce the frame rate or resolution of the movie.

4.2 Real-Time Java Threads

To use regular Java threads, a programmer would construct an object derived from `Thread` class and call `start()` as we show below.

```
MyThread th = new MyThread( . . . );
th.start();
```

By invoking `start()` method, the thread starts execution by calling `run()` method of this `MyThread` object.

Real-time Java threads are created by instantiating a class extended from Thread class called RtThread. `Si`, `Ti`, and `Di` indicate the threads start time, period, and deadline respectively. By specifying a deadline hander, method `meth` will be invoked when it has missed its deadline. This can enable programmers to dynamically adjust attributes of its threads to provide better quality of service for continuous media applications.

```
RtThread rtth = new RtThread(Si, Ti, Di);
rtth.setDeadlineHandler(meth);
```

When constructing a real-time thread object, the virtual machine will actually create a RT-Thread or RTC-Thread. The thread will start interpreting from the `run()` method of this real-time thread object when the `start()` method is called in the Java program.

By mapping Java thread to RT-Thread, Java threads will be scheduled by the real-time scheduler in the kernel. As illustrated in Figure 3a, there is a one to one mapping of real-time Java threads and kernel threads.

When RTC-Thread is used it will be scheduled by the user-level scheduler which is shown in Figure 3b and will not have a one to one mapping between Java threads and RTC-Threads.

In both cases, synchronization between threads is achieved by mutex locks and condition variables which avoids priority inversion problem[13].

4.3 RtThread.class

To use real-time Java threads, we introduced a new class which extends `Thread.class` called `RtThread.class`. Programmers can use the same methods provided in `Thread.class` as well as new methods to support real-time. These are some of the new methods added in RtThread.class.

- `RtThread.setAttr(Time start, Time period, Time deadline)`
  This method will set the timing attributes of the real-time thread.
- `Time RtThread.getStart()`
  This method will get the start time of the real-time thread.
- `Time RtThread.getPeriod()`
  This method will get the period of the real-time thread.
- `Time RtThread.getDeadline()`
  This method will get the Deadline of the real-time thread.
- `RtThread.setDeadlineHandler(method)`
  This method will set the deadline handler.

Using these methods programmers can express aperiodic threads and periodic threads. Aperiodic threads are expressed by specifying deadlines and starts when some external event occurs. Periodic threads can be expressed using start time, period and deadlines. A new instantiation of the periodic thread will be scheduled at the start time specified.

4.4 Programming Example

Here, we will show a simple self stabilizing programming example using real-time Java thread. By self stabilizing we mean that the thread itself can dynamically determine a runnable QoS level under current workload.

In lines 3-6, a periodic Java thread is created with start time set to 3 seconds after current time. (Current time is expressed using java.util.Date()) Period is set to 500 milliseconds and deadline to 400 milliseconds. Time is a class we introduced to express seconds and nanoseconds. In line 7, a deadline handler is set. If the main thread misses its deadline, deadline handler thread will start executing from `deadlineHandler` method. By calling the `start` method, the thread will start executing the `run` method.

In the `run` method of class myRtThread starting at line 17, the periodic thread we have just created will do some job, for example draw a video frame image for a movie application. There is a boolean flag called `missed` which indicates if the thread has missed its deadline previously. If it has not missed the deadline previously (flag is set to false), the period and deadline will be shortened as we show in line 26. The `adjustTimeing` method starting from line 31 is the actual method that will change real-time threads timing attributes. If the main thread has previously missed its deadline, it will just change the value of `missed` flag from true to false.

If the main thread misses the deadline for such reasons as overload in the system, the deadline handler will be invoked. Deadline handler will increase the period and deadline of the main thread hoping that it will not miss its deadline next time and change the missed flag to true as we illustrate in lines 39-45. For the clarity of the code we have made the program very simple, but for a realistic program we can keep a counter and change the period when it has not missed its deadline for a certain amount of time to make it more stable.

By using real-time threads and deadline handlers, programs can dynamically adjust to the changing environment and stabilize to a feasible period and deadline by itself. This simple program illustrates that real-time Java threads and deadline handlers can be an effective tool for providing dynamic QoS control.

```
1. public myProgram{
2.     void main() {
3.        myRtThread rt =
4.           new myRtThread(Time(Date(), 3, 0),
5.                          Time(0, 500000000),
6.                          Time(0, 400000000));
7.       rt.setDeadlineHandler(deadlineHandler);
8.       rt.start();
9.     }
10. }
11.
12. class myRtThread extends RtThread {
13.    int count = 0;
14.    boolean missed = false;
15.    Time period, deadline;
16.
17.   public void run() {
18.      /*
19.       * do some job here
20.       * e.g. drawing a video frame image
21.       */
22.     synchronized (this) {
23.        if (missed) {
24.           missed = false;
25.        }else{
26.           adjustTiming(-10000000);
27.        }
28.      }
29.    }
30.
31.  public synchronized void adjustTiming(int d){
32.     private Time period, deadline;
33.     period = this.getAttrPeriod();
34.     deadline = this.getAttrDeadline();
35.     period.setNsec(period.getNsetc() + d);
36.     deadline.setNsec(deadline.getNsetc() + d);
37.  }
38.
39.  public deadlineHandler(RtThread rt){
40.     System.out.println("missed deadline");
41      rt.adjustTiming(10000000);
42.     synchronized (this) {
43.     missed = true;
44.     }
45.  }
46. }
```

5 Evaluation

We evaluated the performance of real-time Java threads on a Toshiba Dynabook Portege 610 Pentium 90MHz system with 16MB of memory.

We have measured the time elapsed from the start time to the actual time where the Java method was interpreted using our real-time Java thread based on real-time kernel threads (RT-Thread) in Figure 4. We obtained the data from executing a periodic thread 1000 times and getting the average.

It takes 22 microseconds from the clock interrupt

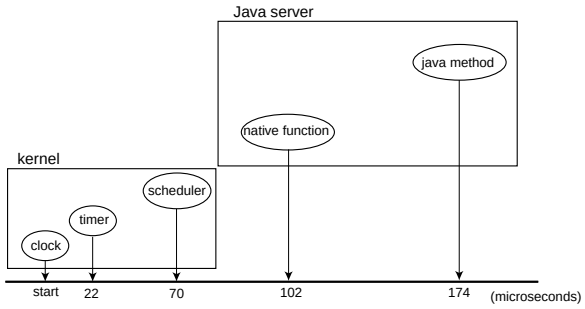


Figure 4. Time Elapsed From Actual Start Time

to the timer in the microkernel. If the timer decides that rescheduling is necessary (the clock interrupt is the start time for a thread), it will notify the scheduler which takes 48 microseconds. For the native function in the Java server to be executed, it takes 102 microseconds from the actual clock interrupt. For the Java method to be interpreted by the real-time thread it needs 72 microseconds inside the virtual machine to look for the methods that it is going to execute. The offset from the actual start time to the time Java method is executed totals to 174 microseconds. We are currently working to reduce this time for periodic threads, where a thread reincarnates every period and looks for the same method that it is going to execute, especially in the virtual machine using smart cache techniques.

In Figure 5 we compared regular Java threads and our real-time Java threads. For regular Java thread, to emulate periodic activity we have created a cyclic thread using `sleep()` method. Inside a `while` loop the thread will sleep for 500 milliseconds. We measured the interval time of the thread entering the head of the while loop.

For real-time Java threads, we created a periodic thread with a period of 500 milliseconds and measured its period. In both cases, cyclic thread or periodic thread will not do any work and we added a number of interfering thread which does random amount of work (with no I/O). All of these interfering threads have the same priority as the periodic or cyclic threads.

We can observe from Figure 5 that as interfering threads increases, regular Java thread behaves more and more unpredictable. Scheduling delay occurs caused from competing threads, and when there are 20 other threads, the delay will increase up to more than 100 milliseconds. Our real-time Java thread will keep its period evne when there are 20 other competing threads. This shows that predictability of real-time Java threads are high with overhead in the system.

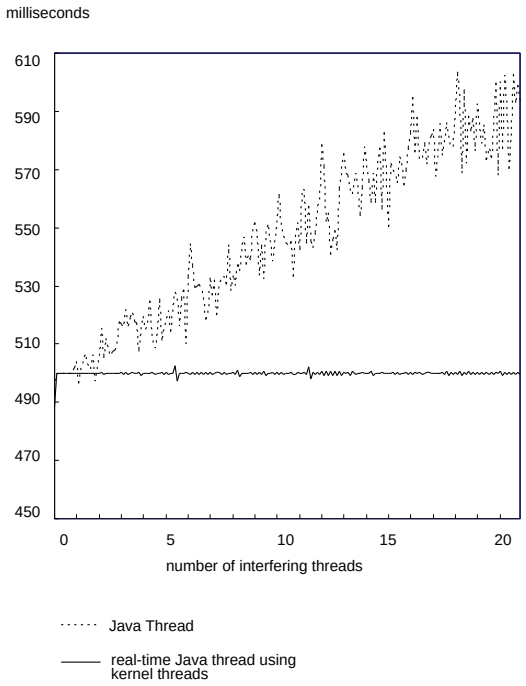


Figure 5. Schedulability of Java Threads and Real-time Java Threads

6 Summary

We discussed the real-time issues in the Java language for developing real-time applications. We also described the architecture of the virtual machine which is implemented on RT-Mach microkernel. The merit of implementing it as a server on RT-Mach is that it can be used in various ways. It can be used from an application on another server or used as an engine for embedded systems. The server supports in memory file system to cache classes as continuous memory blocks to avoid blocking for disk I/O.

We have also added Java threads with real-time attributes. The ability to specify explicit timing constraints is important especially in the way Java is used in distributed environments since the programmer cannot assume the performance of the device which the program is running on. Currently there are two design of real-time Java threads. One based on kernel provided threads (RT-Threads), and the other based on user-level threads (RTC-Thread). Threads are synchronized using priority inheritance protocol to avoid

priority inversion problem.

Experiment data shows the schedulability of real-time Java threads are high compared to regular Java threads even with overhead in the system. We are still improving the performance for periodic threads. Also, enhancements to our virtual machine such as garbage collectors are considered for a more predictable system.

Acknowledgements

We would like to thank Takuro Kitayama, Shuichi Oikawa, Nobuhiko Nishio and all the members of MKng project for their valuable comments.

References

[1] J. Gosling, B. Joy, G. Steele, The Java Language Specification, Addison Wesley, 1996.

[2] Y. Ishikawa,H. Tokuda, and C. Mercer, An Object-Oriented Real-Time Programming Language, *IEEE Computer*, Vol.25, No.10, 1992.

[3] Sun Microsystems, JavaOS(tm): A Standalone Java Environment, *http://java.sun.com*, 1996

[4] H. Tokuda, T. Nakajima, and P. Rao, Real-Time Mach: Towards a Predictable Real-Time System, In *Proceedings of USENIX Mach Workshop*, October 1990.

[5] T. Kitayama, T. Nakajima, and H. Tokuda, RT-IPC: An IPC extension for Real-Time Mach, In *Proceedings of the USENIX Symposium on Microkernel and Other Kernel Architectures*, September 1993.

[6] H. Tokuda and T. Nakajima, Evaluation of Real-Time Synchronization in Real-Time Mach, In *Proceedings of USENIX 2nd Mach Symposium*, October 1991.

[7] T. Nakajima, T. Kitayama and H. Tokuda, Experiments with Real-Time Servers in Real-Time Mach, In *Proceedings of USENIX 3rd Mach Symposium*, 1993.

[8] T. J. Wilkinson & Associates, Kaffe A free virtual machine to run Java code, *http://www.kaffe.org*, 1997

[9] D.Golub, R.Dean, A.Forin, and R.Rashid, Unix as an application program, In *Proceedings of Summer USENIX Conference*, June, 1990.

[10] H. Tokuda and T. Kitayama, Dynamic QOS Control based on Real-Time Threads, In *Proceedings of the 4th Int. Workshop on Network and Operating Systems Support for Digital Audio and Video*, 1993.

[11] S. Oikawa and H. Tokuda, User-Level Real-Time Threads, In *Proceedings of the 11th IEEE Workshop on Real-Time Operating Systems and Software*, May 1994.

[12] S. Oikawa and H. Tokuda, Efficient Timing Management for User-Level Real-Time Threads, In *Proceedings of the 1995 IEEE Real-Time Technology and Applications Symposium*, Chicago May, 1995.

[13] L.Sha, R.Rajkumar, and J. P.Lehoczky, Priority inheritance protocols: An approach to real-time synchronization, *IEEE Transactions on Computers*, Vol39, No.9, September 1990.

[14] Kelvin Nilsen, Java for Real-Time, *Real-Time Systems Journal*, Vol. 11, No. 2, 1996

[15] E. C. Cooper, and R. P. Draves, C threads, *Technical report, Computer Science Department, Carnegie Mellon University*, CMU-CS-88-154, March, 1987.