Java for Real–time Object–oriented Programming

Antonio Boccalatte and Mauro Coccoli

University of Genova
Department of Computer, Communication, and Systems Science
via Opera Pia, 13–16145 GENOVA
nino@dist.unige.it, coccoli@dist.unige.it

Abstract

Real–time programming is simply a necessity in many fields of application and specific real–time operating systems exist in order to fulfill the necessity of users to have time determinism in their applications. Anyway, often time constraints are not so hard and it is possible to identify hard and soft real–time applications, but time determinism keeps on being the mandatory requirement. The aim of this paper is that of describing the results of an activity devoted to investigate the possibility of real–time programming in a Java environment. A package has been developed in order to give a Java platform real–time capabilities of scheduling threads with time–based policies or by priority and events. A set of methods for threads to communicate through queues, messages, and mailboxes has been developed too.

1 Introduction

It can be claimed that *"Java is not an appropriate language for real–time programming"*. In fact, given general characteristics that a real–time system should offer [1], Java seems not to be adequate at all for real–time programming; the main motivation can be found in the fact that Java is a platform–independent language, and a real–time scheduling of Java Threads can not be guaranteed if there is no *a priori* knowledge about the operating system scheduling characteristics. Moreover Java has an automatic garbage collector whose influence should be taken in account (see [2], [3]) . Anyway, since Java naturally supports *Threads* [4], it appears to be suited for the experiment proposed in this paper. In fact, real–time applications have to be written as a series of separate component programs that can execute concurrently in a multi threading organization; every Java thread is a complete program capable of independent execution sharing its memory with others, always keeping separate addresses so that context switching can be very fast. Moreover, since its first appearance, in May of 1995, Java was presented as a language for facilitating the development of embedded systems software [5], and most embedded computer systems, have to deal with real–time constraints. Generally speaking, it is possible to distinguish between *hard* and *soft* real–time constraints and it can be seen that it is possible to have a certain degree of real–time reliability with Java too, when Java programming is performed in a *special* environment. In order to provide Java with real–time characteristics, a first solution is being developed by a consortium comprehending IBM, Hewlett Packard, and Newmonics, that has already developed a modified Java Virtual Machine, called *Perc* (a real–time dialect of Java). It is important to notice that such a consortium, however supported by major software houses, is not sustained by Sun, that has made up an Expert Group on its own. Whomever it can derive from, it is evident that such a solution would have a great impact on the platform independence that a Java application should keep on claiming and guaranteeing. Another possible, alternative, solution, should go towards the development of specific real–time programming facilities for handling process scheduling and real–time features. In this paper, this second way, has been followed, and a package for Java programming has been developed, that allows to respect time constraints that (soft) real–time applications always have to deal with.

After this short introduction, the paper will be organized as follows: in Section 2 basics on real–time programming will be reported, that have driven the realization of the proposed package; in Section 3 motivations for investigating in the field of Java real–time (JRT) programming are presented, while in Section 4 the Java Virtual Machine (JVM) is discussed, motivated by the possibility of performing a slight modification to its architecture in order to build a

modified JVM for JRT. The Java package is described in Section 5 and the results of the performed tests are finally presented and discussed in Section 6. General considerations and conclusions follow.

2 Real–time Programming

Whenever a real–time operating system is not present, it is up to the programmer to give its tasks pseudo real–time characteristics, if it is necessary. The following attributes can be considered mandatory requirements for a system to be able to deal with real–time: multitasking/multithreading, priority management, preemption, synchronization mechanism, timing.

Such characteristics can be guaranteed by the presence of specific software components: task manager, memory manager, message queue, event and asynchronous signal, semaphore, timer, interrupt handling, error manager [6].

Generally speaking, real–time applications need concurrent execution of several tasks, sometimes one depending from other ones. This is multi tasking programming, where each task is an executable portion of code with a specific role with its own memory area, stack, stack pointer, and program counter. Switching among concurrent tasks is driven by interrupts causing suspension of processes. Suspending a process always has to be a safe operation, so registers, program counter, and more (the *context*) have to be saved, while the context of the next executing task has to be loaded. Context switching is up to the operating systems' scheduler.

Real–time processes can be represented as if they could be in one among four possible states:

1. Active–The process is in execution.
2. Ready–The task is ready to be executed but the CPU is busy.
3. Blocked–The task cannot be executed due to a missing resource.
4. Dormant–Either if the task has finished its job or the task has not yet requested CPU time.

The possible transitions are shown in the following sketch, (Fig. 1).

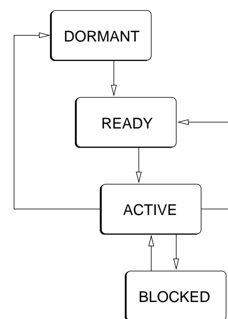

Figure 1. The possible states a task can be in (operating system view)

3 Why Java for Real–time?

The starting point for the answer to the above question is the Java language claimed platform independence. For real–time and embedded systems' development where, in most cases, using custom hardware is the unique solution, having the possibility to write general purpose software that then will run everywhere, is a more than fascinating item. Moreover, many embedded systems' applications have more and more the necessity of Web connection [7] and Java is, since the start, the best language for Web programming because it has a special ability for the development of distributed applications, and it offers high–level graphical user interfaces with a very small programming effort.

In particular modern Internet appliances require Internet connectivity, security, interactivity, and dynamic–extensibility features: Java makes all of these available.

From a software technology point of view, Java is a well suited development tool for the embedded market for the following features it can guarantee:

- security (no memory pointers, sand box model);
- portability (object–oriented programming enable and promote code reuse, ported to many real–time OS' and processors);
- small memory sizes (fits in small ROM footprints, Java byte code format smaller than native code, dynamic loading/unloading when needed);
- dynamic (allows new features/patched to be loaded, secure).

Most applications designed to run on an embedded system need real–time features in order to deal with the real world via I/O devices and to be able to respond with determinism to the operating system messages and alarms. Moreover, often time constraints have to be strictly respected. Java language supports multi threading programming, through synchronization modifiers introduced in the language syntax, through specific classes developed for multi threading which can be inherited by other classes, and by recovering data areas that are no longer being used for multiple threads, that is garbage collection. Multi–threading is in the *pro*s column from the real–time point of view, garbage collection, on the other hand, is opposite to any real–time behaviours, due to its time–unpredictability. Java can not

act direct memory access and in such a way reading from a register or writing a device driver is not an easy task.

4 Inside the Java Virtual Machine

Inside the Java Virtual Machine, each thread can be in one among the four possible following states:

1. Initial–Any thread is in this state from the moment it is created, until the method `start()` is called.
2. Runnable–Once the `start()` method has been called.
3. Blocked–If waiting for an unblocking event.
4. Exiting–After the `run()` method has terminated its execution or the `stop()` method has been called.

Only one thread at a time can be executed but several threads can be in the Runnable state at the same time. In such a condition, it is up to the JVM to select which one of the possible threads will jump in execution. Such a choice is not a random decision but it must be driven by a preemptive process scheduler following a priority–based scheduling policy. Every thread has its own priority, ranging from 1 up to 10; the value 5 is the default one. Once assigned a different priority to a specific thread, such value can not be any more modified by the JVM. Always, the thread with the highest priority will be executed. If a thread is running and another thread with higher priority reaches the Runnable state, this last one will jump in execution (that is preemption) while the former will keep on being a Runnable task but it will not go on further until the other thread has been executed.

Such a policy may seem to show a predictable behaviour. Indeed this is true when no two threads have the same priority. Since priority values are chosen among ten, complex systems having more than ten Runnable threads could not be deterministic any more. How does the JVM select the thread to run among the same priority ones? The JVM keeps track references to threads in thirteen linked lists (Fig. 2): one list for processes in each of the state Blocked, Initial, Exiting, that is three lists; the remaining ten lists are relevant to any priority value of threads in the Runnable state. Typically, the higher priority task is promoted but this is only one among many possible criteria by which the JVM sets the thread to be executed. Whenever more than one process have the same priority, the operating system dependence arises. The running task can change its state depending on one of the following events.

1. The thread is awaiting for a particular condition or has finished its job. JVM will retrieve the first Runnable thread within the list. It is the highest priority Runnable thread.

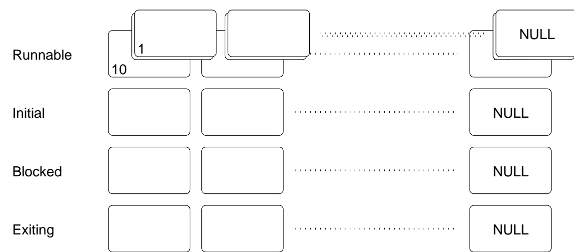

Figure 2. The linked lists inside JVM

2. A thread with higher priority than the running one gets in the Runnable state. This depends on the preemptiveness of the JVM.
3. The CPU time–slice is over for that thread. Any process is assigned a maximum time for CPU usage and it has to be released after this time has passed.

This last item is connected to the operating system underlying the Java environment. Many UNIX operating systems do not have such an event in such a way that scheduling can be said to be deterministic for it is based on the first two events occurring only. This is not the case of the Microsoft Windows operating systems and there can be a non–deterministic scheduling of threads, depending on the events driven by time–slice.

5 The Real–time Package

In this Section, the developed real–time package is described; a short overview of the classes and methods it offers the programmer is presented.

5.1 Java Timing

When working with Java, there exists one only way to deal with time, the *real* time, that is the `System.currentTimeMillis()` method; when invoked, it gives out the elapsed time since 1970, in milliseconds. Another time–based method is `sleep()` able to suspend a thread for a specified number of milliseconds. Anyway, this last method can not be considered reliable, since its result depends on the platform the software is running on.

A possible solution for the realization of periodic threads (e.g., a control loop) is an infinite looping of a process suspending itself by a `sleep()` call but such a control loop will be affected by an error that, in some cases, could be not tolerable.

5.2 The Classes in the Package

The developed package extends (by the Java meaning of extension) the functionality of the basic `Thread` class by adding new methods in order to be able to manage semaphores, messages, and events. The new classes are listed in the following:

- `McQueue.java`– A class whose methods allow to access the data structure which has in charge the management of suspended processes. The policy this queue is based on in this first implementation is a FIFO technique.
- `McError.java`– A class for managing errors.
- `McEventActivation.java`– A class which extends the `Exception` class since any event can be regarded as en exception to be handled and processed through specified methods. In this case the extension gives the possibility to immediately activate a process despite whatever is running.
- `McEventManagement.java`– A class containing the method to access to whom manages the event of the type `McEventActivation`.
- `McJob.java`– A class which extends the class `Thread`, whose methods allow to interact with other data inside the package and to access the synchronization primitives.
- `McClock.java`– A class which creates a process with a fixed dead–line, extension of the `McJob` class.
- `McMbox.java`– A class which defines characteristics and methods of a mailbox for the processes.
- `McMboxList.java`– A class whose methods allow to manage the mailboxes of single processes.
- `McSemaphoreList.java`– A class whose methods allow to manage the semaphores of single processes.
- `McMessage.java`– A class to define messages for interprocess communication.
- `McNode.java`– A class to define the characteristics of suspended processes.
- `McProcess.java`– A class to keep track of all of the processes.
- `McSemaphore.java`– A class to define a semaphore characteristics and methods. Any process standing on a semaphore is registered in the queue of suspended processes.
- `McState.java`– A class container for all of the informations regarding created semaphores.
- `McTask.java`– A class with all the needed methods to interact with data and with all of the synchronization primitives.

By using such classes, it is possible to create programs to manage processes for real–time applications, based on a driven scheduling and on synchronization primitives.

5.3 A Real–time Process Scheduler

A process scheduler has been written, based on the use of the previously described classes. It is a trial for verifying the effective Java real–time behaviour in such a situation and the limits of the presented package. From a trivial graphical user interface the following tasks can be performed:

- view the queue of suspended processes
- choose a prespecified scheduling algorithm
- create a process
- activate a known process
- define the time–slice
- modify process parameters (i.e., priority)
- view a draft scheduling output

Scheduling can be manual or automatic, where, for the sake of simplicity, here automatic means priority driven FIFO. Manual scheduling provides the programmer with the ability to activate a process among the ones in a process list identified by a unique tag. A process may have been created as a periodic one or not. In the priority driven scheduling, any newly created process will be scheduled according to its time demand and priority. A suited `McEventList` manages all the processes and any events yet to come is called `McNode`. Any process has its own method `McControl()` whose aim is to define the parameters for the next event associated to the running task. Once defined such parameters, an update is performed before calling the `McSuspend()` method. Events are arranged by a time–based order and in case of concurrency, then priority will drive the choice.

In order to define the processes that will have to run, it is possible to extend the `Job` class overriding the `run()` method, and adding methods for synchronization and communication while the code to be executed is linked by a method `control()`.

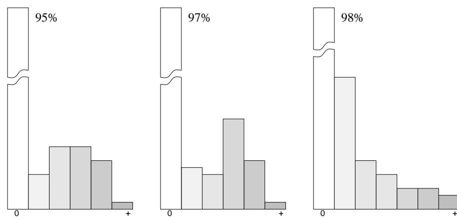

Figure 3. Timing errors on a periodic task along 5, 10, and 20 [min] of execution (Linux)

6 Benchmark and Experimental Results

Different tests have been performed on periodic tasks relevant the respect of their given timing. Some meaningful discrepancies in the behaviour of real–time tasks have been encountered depending on the underlying operating system: the most widely used PC operating systems have been chosen for testing, that is Linux and Microsoft WindowsNT (due to the existing in–laboratory configuration, the Linux machine had a worse processor than the WindowsNT one but the tasks were not heavy computing applications). A periodic task has been run for a duration of 5, 10, and 20 [min] in order to evaluate time reliability. Such a task should have been activated at exact time intervals (e.g., a sampling period in a control system). Results are shown in bar–diagrams which report the number of times the task has been activated at the right time instant (marked with 0), or late (marked with +) at the right side of the graphic. Another test has been carried on by launching 3 real–time threads at the same time, to be executed in parallel, whose periods were 100 [ms], 500 [ms], and 1000 [ms]. Such a test has been performed along a 10 [min] duration: it aimed to evaluate the time accuracy in the execution of the three tasks. Bar–diagrams report as in the previous experiment delays on the tasks execution.

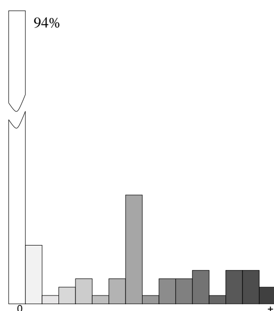

Figure 4. Timing errors on the 100 [ms] periodic task (Linux)

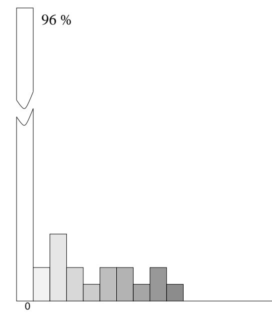

Figure 5. Timing errors on the 500 [ms] periodic task (Linux)

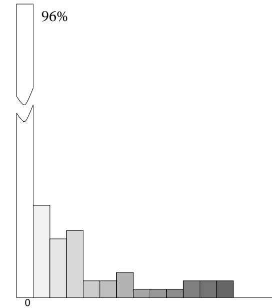

Figure 6. Timing errors on the 1000 [ms] periodic task (Linux)

6.1 Results on a Linux PC

Running the first of the above experiments on a Linux PC (obviously without RT extensions), the results shown in Fig. 3 have been obtained. They show a hit ratio for timing of the periodic thread of 95%, 97%, and 98% for different lasting times of the test. It can be noticed how, as time grows, the accuracy becomes higher and higher.

For the second experiment too, bar–diagrams are reported, (Fig. 4, up to Fig. 6) relevant to the three tasks with different timing, and the results are 94% hit for the 100 [ms] thread, 96% for the 500 [ms] one, and 96% for the 1000 [ms] thread.

6.2 Results on a Windows NT PC

The same bar–diagrams for the above experiments are reported in the case when real–time threads have been run in a WindowsNT environment. Results are shown in Fig. 7, up to Fig. 10. Hit percentages are not reported but diagrams are in scale (anyway the zero–delayed are about 45%).

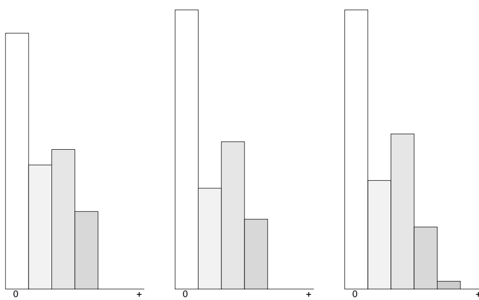

Figure 7. Timing errors on a periodic task along 5, 10, and 20 [min] of execution (WindowsNT)

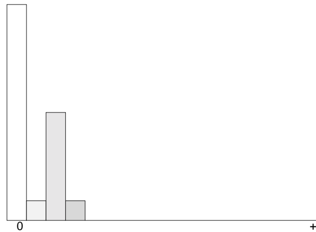

Figure 8. Timing errors on the 100 [ms] periodic task (WindowsNT)

7 Conclusions

The problem of real–time programming in Java has been considered, and a real–time package has been developed in order to give the Java language some real–time reliability. General problems of this language for such applications have been identified, and strong motivations for trying to provide Java with real–time programming capabilities have been given. System primitives have been implemented to give the programmer the ability to deal with binary semaphores, events, messages, process control, memory sharing. Considering the restricted area of soft real–time applications where the level of urgency of every task is specified and the CPU is assigned to the most urgent thread that is ready to execute, the other aspects of real–time programming holding, the package that has been presented in this paper can be profitably used.

An interesting outcome of this work can be considered the observed operating system sensitivity of the package. The proposed benchmark has outlined a better behaviour of the Linux PC with respect to the WindowsNT one if we consider the average error. On the other hand, if we consider the absolute error, the WindowsNT PC has shown narrower diagrams suggesting that the timing is not respected in the most cases but the error is very small while in the Linux PC,

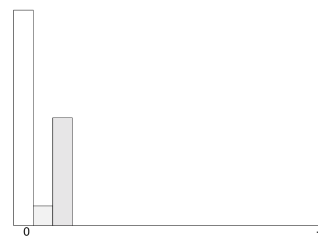

Figure 9. Timing errors on the 500 [ms] periodic task (WindowsNT)

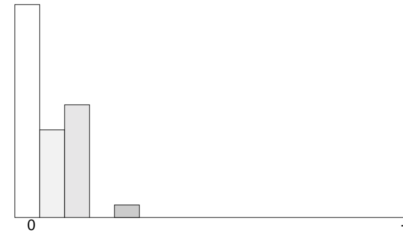

Figure 10. Timing errors on the 1000 [ms] periodic task (WindowsNT)

sometimes timing is wrong for a great amount of [ms]. Depending on the specific application the system is intended to, one behaviour could be better than the other one. Much attention has to be paid in the choice of the priority of the threads to be run.

References

[1] N. Nissanke. *Real–time Systems*. Prentice Hall, 1997.

[2] R. Johnson. Reducing the latency of a real–time garbage collector. *ACM Letters on Programming Languages and Systems*, pages 46–58, March 1992.

[3] H.G. Baker, Jr. The treadmill: Real–time garbage collection without motion sickness. *OOPSLA Workshop on Garbage Collection in Object–oriented Systems*, 1991.

[4] The java language overview. Sun Microsystems, Inc.: Mountain View, CA, 1995.

[5] The java language environment: a white paper. Sun Microsystems, Inc.: Mountain View, CA, 1995.

[6] D. Ripps. *An Implementation Guide to Real–time Programming*. Yourdon Press Computing Series, Prentice Hall, 1989.

[7] V. Perrier. Can java fly? adapting java to embedded development. *Embedded Developers Journal*, pages 8–18, September 1999.