Fault Tolerance with Real-Time Java

Damien Masson, Serge Midonnet

Fault Tolerance with Real-Time Java

Damien Masson[1] and Serge Midonnet[1]

[1]Université de Marne-la-vallée
Institut Gaspard-Monge
Champs-sur-Marne, France
{damien.masson, serge.midonnet}@univ-mlv.fr

Abstract

After having drawn up a state of the art on the theoretical feasibility of a system of periodic tasks scheduled by a preemptive algorithm at fixed priorities, we show in this article that temporal faults can occur all the same within a theoretically feasible system, that these faults can lead to a failure of the system and that we can use the data calculated during control of admission to install detectors of faults and to define a factor of tolerance. We show then the results obtained on a system of periodic tasks coded with Java Real-Time and carried out with the virtual machine jRate. *These results show that the installation of the detectors and the tolerance to the faults makes an improvement of the behavior of the system in the presence of faults.*

1. Introduction

Real-Time systems occupy an increasingly significant place in industry and are present in an increasing number of fields. Today, because of temporal constraints, real-time programmers have to use low-level programming languages like the assembler or the language C. They cannot benefit from the advantages of the Java language: a high level of abstraction (oriented object approach) and the independence from the execution platforms (portability). This led in 2001 to the Real-Time Specification for Java (*RTSJ*)[4]. This specification describes a set of classes and interfaces and defines the constraints imposed for real-time virtual machines (memory management, scheduling algorithms and synchronisation mechanisms). The work on this specification is still in progress, as attested by the publication in June 2005 of its version 1.0.1. There are also many *RTSJ* virtual machines. Some are commercialized as *Jamaïca VM* from *AICAS* society. Some others exist for research as *RI* which stand for *reference implementation*. The latter was made to prove the feasibility of real-time Java adventure. Eventually *jRate* [7, 6], an open source project based upon an extension of the *GNU GCJ compiler* or *flex* [8] developed in *MIT*. Recently, *Sun Microsystems* themselves announced their own implementation of the specification called *project Mackinac* [13, 3].

If this specification offers to programmers methods to compute an admission control for real-time periodic tasks, the tested machines do not offer a valid implementation. We can easily show a non feasible set of tasks for which *RI* returns *feasible*, and we can see in the file `PriorityScheduler.java` that feasibility methods are not yet implemented in *jRate*.

So, to begin this work, we have drawn up a state of the art on the feasibility analysis issues for a fixed priority scheduled periodic task system. This study leads us to write the deficient methods of *RI* and missing ones in *jRate*. However, though these methods give us a theoretical feasibility of the task system, in practice faults can occur for many reasons. To take into account these faults, an approach usually met in the literature is to install overload detection and treatment mechanisms [12, 9, 5]. Our approach, in the other hand, consisted on using an admission control with fault detection and treatments mechanisms, in order to avoid any overload.

We start by presenting the admission control algorithm in Section 2. In Section 3 we define the temporal faults and we explain how we can detect them by overloading methods of the *Real-time Specification for Java (RTSJ)*. Section 4 presents the various treatments of the faults we have established. We explain which mechanisms of measurement we

1-4244-0054-6/06/$20.00

set up in Section 5 and have their results in Section 6. Finally we present in conclusion the prospects for work offered by these results.

2. Control of admission

The theory of scheduling applied to real-time systems have been largely studied over the past twenty years and many results were published [11, 10, 1, 2]. We limited our study to a fixed priority pre-emptive scheduling algorithm because all the *RTSJ* implementations have to offer it at least. A task, denoted by τ_i has a cost C_i, a relative deadline D_i, a period T_i and a priority P_i.

In this section, we present the results described by Liu and Layland in [11] and generalised by Lehoczky in [10] to the tasks with the deadline higher than the period.

2.1. Load test

The system load, denoted by U, can be calculated by:

$$U = \sum_{i=1}^{N} \frac{C_i}{T_i} \quad (1)$$

There are two possibilities:

- if $U > 1$, the system is not feasible,
- else the load condition is not enough to conclude.

2.2. Worst case response times computation

	P_i	D_i	T_i	C_i
τ_1	20	6	6	3
τ_2	15	2	4	2

Table 1. System task data

The response time for a task job is the time interval between its activation and its termination. The worst case response time for a task is the response time of its longest job in the worst situation. In the particular situation where $D_i < T_i$, the worst situation is reached when all the tasks are activated simultaneously. As shown in table 1 and figure 1, this is not true in the general case. Consequently, the calculation of these worst case response times in the general case requires a thorough study.

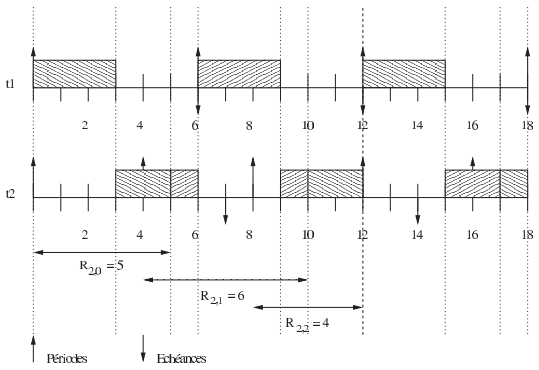

Figure 1. Response times

```
[Return the worst case response time in nanosecondes]
Function WCResponseTime( S : Schedulable) : long integer
    R_q : long integer
    R_max : long integer ← 0
    q : integer ← 0
    R_tmp : long integer
    While ( true) do
        R_q ← 0
        R_tmp ← S.cost * (q + 1)
        While (R_q ≠ R_tmp) do
            HP(S) : Set of Schedulables with higher or equal priority than S priority
            R_q ← R_tmp
            R_tmp ← S.cost * (q + 1)
            For all S_j in HP(S) do
                R_tmp ← R_tmp + ⌈R_q / S_j.period⌉ * S_j.cost
            done
        done
        R_q ← R_tmp
        If (R_max < R_q − q * S.period) then
            R_max ← R_q − q * S.period
        end If
        If (R_q ≤ (q + 1) * S.period) then
            break;
        end If
        q ← q + 1
    done
    return R_max
End
```

Figure 2. Worst case response time computation of a task in a fixed priority preemptive system

Many works were carried out on the calculation of these response times. They led to the algorithm shown

on Figure 2.

The algorithm is based on the following observations: a job activation can only be delayed by a job of a higher priority task or by the previous jobs of the same task. Furthermore a job can only be preempted by a higher priority task job. A recurrence formula enables us to compute the response time of each task job from the response time of the previous job. We just have to iterate this computation over all the jobs until we find a job with a response time smaller than its period. In that case, there is no influence of the job on the following jobs: the worst case response time is then the maximum response time computed during the iterations.

2.3. Implementation

We offer to developers a new package called `javax.realtime.extended`. This package contains the class `RealtimeThreadExtended` which extends `RealtimeThread`.

The methods concerned by the job control are `addToFeasibility()` and `removeFromFeasibility()` and are overloaded to delegate the work to one of our classes called `FeasibilityAnalysis`. This class implements the algorithm of Figure 2.

3. Fault detection

A system failure occurs when one (or many) task has a job which missed its deadline (D_i). If we always perform an admission control before starting a task system, it should never have any system failure. However, algorithms shown up on the previous section lead on the hypothesis of the exactitude and the respect of tasks parameters. In particular, the task jobs have always to respect their costs (C_i). Since this cost is often obtained by a statistical work, it is not reliable. So it is possible a task job takes a little bit more than its cost, either because it was underestimated, or because of an external event with the system. This is called a fault. A fault can lead to the failure of the faulty task or the failure of a task with a smaller priority.

The detection of a cost overrun is a difficult problem, because it implies being able to know at any time how much CPU resource a job has already consumed. However, the worst case response time computation carried out during the admission control gives us for each task job a date, depending on its activation, before which it has to be finished.

A worst case response time overrun implies a cost overrun.

All the critical instants to watch depend on the jobs activations. This is why a detector can be a periodic task, with a period equal to the task period and with an offset equal to the task worst case response time. This periodic approach enables us to avoid the creation of an instance of a detector for each job. This decreases the cost overrun induced in the system by our detectors since there is only need for one real-time task for each thread.

3.1. Implementation

In `RealtimeThreadExtended`, we overload the method `start()`. Our method starts a periodic detector with an offset equal to the worst case response time just after having called the method `start()` of the super-class. The detector is an instance of `PeriodicTimer` which checks the states of a boolean value and a job counter. This job counter was added to our real-time thread class. If the job is not finished when the timer is called, a fault treatment can be started. The states of the boolean value and the counter are up-to-date in overloading the method `waitForNextPeriod()` which is called at the end of each periodic job and which is blocking until the next job activation. Our method becomes:

```
public boolean waitForNextPeriod(){
  computeAfterPeriodic();
  boolean returnValue =
    super.waitForNextPeriod();
  computeBeforePeriodic();
  return returnValue;
}
```

With this approach, we are able to add actions just before and just after each job.

It implies that the method `waitForNextPeriod()` has to be called before the first job, which is often the case in order to minimise the influence of the method `start()` call.

4. Treatments

We are now able to detect some kinds of faults in a real-time system. How can we treat these faults ? First, we have to look at the consequences of a worst case response time overrun:

- the system is still feasible, there are no more consequences than a little offset on the worst case response times of the tasks with a lower priority,

- the faulty task misses its deadline and probably provokes the failure of tasks with lower priority,
- the faulty task ends before its deadline but the offset leads to failure for some tasks with lower priority.

Our goal is to prevent that the faulty tasks with a strong priority cause the failure of non-faulty tasks with a lower priority. To do so, we considered three treatments presented in the following subsections. We compare these treatments and we measure the improvement of the system behavior in a practical way on an example of three tasks in Section 6.

4.1. Instantaneous stop of the faulty tasks

The first idea of treatment consists in simply stopping the faulty tasks. This approach is very pessimistic, because making a fault can have no consequence on the system feasibility.

Let us note in addition that, in Java, it is not possible to stop a thread brutally. The theoretical solution would consist in lowering the priority of the task so that this one can be preempted, but this possibility is not yet implemented with *jRate*. For the needs of our analysis, we thus added a boolean field in the class `RealtimeThreadExtended` which is checked after each instruction of the loop constituting the periodic treatment. If this boolean gets `true`, then the loop is broken and the thread is stopped. That causes an annoying side effect: in the method `run()`, the thread must check the state of the boolean, and thus calls the method `RealtimeThread.currentRealtimeThread()`, the cost of which is not bounded. Consequently, the task will regularly make small cost overruns, about a few milliseconds. These cost overruns remaining lower than the precision of our detectors, that does not obstruct our analysis.

4.2. Stop after an allowance factor granted to the task

In order to have a less pessimistic approach, and to make it possible for the faulty tasks to continue as long as they can without obstructing the lower priority tasks, we have to calculate what we will call the allowance of the system, *i.e.* the additional cost we can grant to each task while keeping a theoretically feasible system.

To compute this tolerance, we carry out a binary search of the maximum value which can be added to the costs of all the tasks so that the system remains feasible (within the meaning of the analysis described in the previous section).

During this computation, we also compute new worst case response times which take into account the allowance. The tasks will now be stopped after these new worst case response times.

4.3. Stop after an allowance factor granted to the system

The treatment previously described considers that all the tasks have the same probabilities to make faults. The free time in the system is equitably distributed between all the tasks.

In the last treatment we implemented we consider that the higher the task priority is, the more it has the right to make a fault: this time, a task is stopped after a worst case response time overrun equal to the maximum free time available in the system.

This allowance is obtained by seeking the maximum value which can be added to each task cost.

If the first faulty task finishes before having consumed all its allowance, the remainder is allocated to the other faulty tasks. A task allowance is obtained looking for the maximum cost overrun this task can do and substracting the more priority tasks overrun.

5. Measurements

In order to have measurements, we developed various tools. The first one enables us to parse a file which describes the tasks in the system. It builds and runs the tasks automatically. During the system execution, many data are collected. A second tool provides a chart of these data in the form of a time series chart.

The data collected, which are the key dates in the system life, are:

- the beginning of a job (the moment when the method `computeBeforePeriodic()` is called),
- the end of a job (the moment when the method `computeAfterPeriodic()` is called),
- the release of a detector.

In order to have a maximum precision, we use for these measurements an Intel processors characteristic: these processors have an instruction `RDTSC` (*ReaD*

Time-Stamp Counter) providing the number of cycles carried out since the start-up of the system.

We thus wrote a library in Java native interface (*JNI*) allowing us to exploit this instruction, in order to obtain durations with a nanosecond precision.

We write these times in `StringBuffer` fields in order not to slow down the system with in-out operations. At the end of the execution of the system, these fields are written in a log file which can then be interpreted by our tool of time series chart.

Figures presented in Section 6 were obtained thanks to these tools, and thus are experimental results obtained on true Real-time Java systems of tasks, and not the awaited theoretical behaviors.

6. Results

We present in this section the results obtained on the system of tasks described in table 2. A cost overrun was voluntarily added for the priority task, which represents the most unfavourable case. We will compare the time series charts obtained in the following cases:

- no detection mechanism,
- active detectors, no fault treatment,
- active detectors, immediate stop of faulty tasks,
- active detectors, allowance granted equitably to all tasks,
- active detectors, allowance granted totally for the first faulty task.

	P_i	T_i	D_i	C_i	$WCRT_i$	A_i
τ_1	20	200	70	29	29	11
τ_2	18	250	120	29	58	11
τ_3	16	1500	120	29	87	11

Table 2. Tested tasks system

Figures from 3 to 7 are obtained with the tool that we developed, and present the behavior of the system during the fifth job of task τ_1, which coincides with the activation of a job of τ_2 and τ_3.

On these figures $\uparrow$ represents the periods and $\downarrow$ the deadlines. The $\updownarrow$ materializes the detectors, and $>$ indicates the worst case response times.

For our tests, we used the *RTSJ jRate* compiler because it is the only one among the tested virtual machines (*RI*, *JamaïcaVM*, ...) to have a good implementation of some essential aspects for our work, such as the signature of the method `RealtimeThread.waitForNextPeriod()`. We used a 2 GHz processor INTEL Pentium 4 machine with 500 MB of memory and a real-time kernel Timesys *2.4.18-timesys-4.0.243*.

6.1. Execution without detection

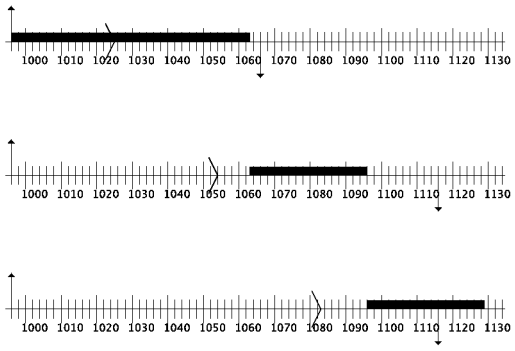

Figure 3. Execution without detection

Figure 3 shows us that task τ_1 makes a temporal fault at moment $1020m$. It ends before its deadline, just as task τ_2 but task τ_3 misses its deadline. It is the case we wish to avoid.

6.2. Execution with detection, without treatments

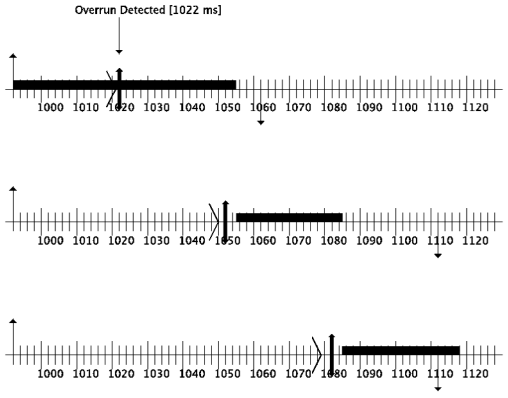

Figure 4. Execution with detection, without treatments

The execution presented in Figure 4 is similar to the one presented on Figure 3. One can notice that the

detectors have a small delay. This delay cannot exceed ten milliseconds. This is due to the implementation of the `PeriodicTimer` in *jRate*: if the value given for the first release is not a multiple of ten, the precision is not good. We thus voluntarily round the release values of the detectors. This is why the detector of task τ_1 has a $30 - 29 = 1$ millisecond delay, that of τ_2 $60 - 58 = 2$ milliseconds and that of τ_3 $90 - 87 = 3$ milliseconds, which can be checked on the diagram.

The overrun generated in the system by the presence of the detection mechnism is that of a pre-emption, in addition to an unbounded value which corresponds to the boolean testing/test, as explained in Section 4.1. One can estimate this overrun to be inferior to the underlying accuracy of the real-time machine, hence to be negligible. Yet, one has to bear in mind that the more tasks in the system, the more sensors, hence, the higher the influence of this overrun on its execution.

6.3. Instantaneous stop of the faulty tasks

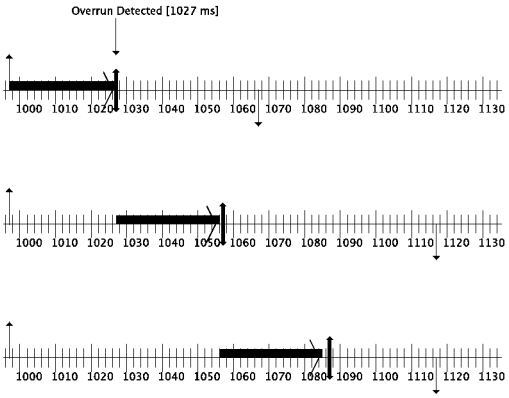

Figure 5. Execution without allowance

In the execution presented on Figure 5, the tasks are stopped as soon as they make faults. One can notice that the only task to miss its deadline is task τ_1. However after the end of the execution of task τ_3, the processor is free, and there remains time before its expiry. We can then think that task τ_1 could have time to finish, or at least to be carried out at greater length without causing the failure of τ_2 and/or τ_3.

6.4. Allowance granted equitably to all tasks

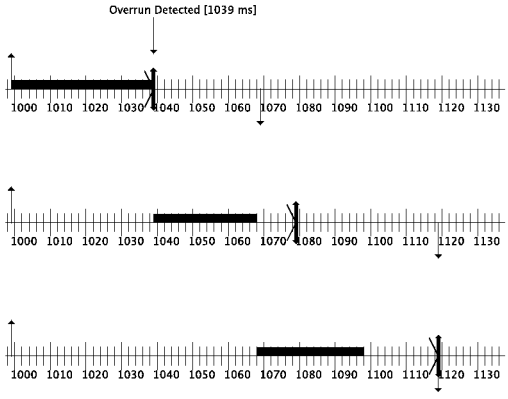

Figure 6. Allowance granted equitably to all tasks

Figure 6 results from an execution in which we left all the tasks the same allowance: eleven milliseconds. The worst case response times indicated by $>$ now represent the worst response times by taking into account a cost overrun of the task equal to eleven milliseconds.

Task	worst case response time with cost overruns
τ_1	$WCRT_1 + A_1 = WCRT_1 + 11$
τ_2	$WCRT_2 + A_1 + A_2 = WCRT_2 + 22$
τ_2	$WCRT_3 + A_1 + A_2 + A_3 = WCRT_3 + 33$

Table 3. Worst case response time with cost overruns

Table 3 gives these new worst case response times. We note that only task τ_1 is stopped, and that it had more time to be carried out than in the previous case. However, we also notice that there remains unused CPU time, because tasks τ_2 and τ_3 did not consume their allowance, since they did not make cost overruns.

6.5. Allowance granted totally to the first faulty task

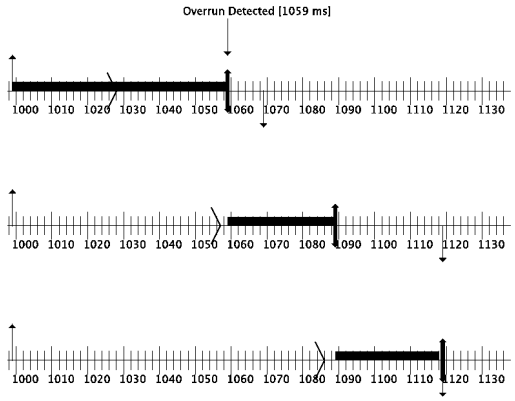

Figure 7. Allowance granted totally to the first faulty task

Figure 7 corresponds to the last treatment considered: all the system time available in the worst execution case, that is to say thirty three milliseconds, is granted to the first faulty task. If this task does not consume all this allowance, and if another task makes a fault, it will benefit of what remains. Task τ_1 is thus stopped thirty-three milliseconds after its worst case response time and as neither τ_2 nor τ_3 make fault, they both finish just before their deadlines.

7. Conclusions and future works

The study of the theoretical results of the analysis of feasibility of a system enabled us to provide a correct establishment of the methods specified by the standard *RTSJ* for control of admission, which we could concretely check on the machine *jRate*.

We saw that theoretical feasibility is based on data such as the cost of the tasks which can possibly be exceeded during a real execution because of external factors, causing faults within the tasks system.

Thanks to the study of the worst case response times, we could define a type of fault whose detection does not require a complete monitoring of the CPU usage. We deduced a mechanism of detection which we coded and tested.

Finally we showed that the theoretical study made it possible to obtain a factor of allowance which maximizes the execution time of a task before its fault causes a failure of the system.

However, our study considers a rather static system, in which all the tasks are known before launching, making possible to set up expensive algorithms in time to compute the response times and the allowance of the system.

Our objective in the continuation of this work will be to reach the same results in a more dynamic system where tasks can be added or removed "in real-time" by adapting the behavior of our detectors.

Moreover, if the cost of a task can be underestimated, it is also possible to overestimate it. Consequently, we can consider to dynamically study the system in order to detect these costs under-run and to reassign resources for faulty tasks.

Besides, we have considered neither the issues related to precedence constraints nor the ones deriving from the share of resources among the various tasks of the system. In the latter case, it would be advisable to study the influence of tolerance on the determination of the blocking time (b_i).

Another main line of our research will consist in studying the faults detection and tolerance in the case of aperiodic tasks.

References

[1] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings. Hard Real-Time Scheduling: The Deadline Monotonic Approach. In *Proceedings 8th IEEE Workshop on Real-Time Operating Systems and Software*, Atalanta, 1991.

[2] E. Bini, G. C. Buttazzo, and G. M. Buttazzo. Rate monotonic analysis: The hyperbolic bound. *IEEE TRANSACTIONS ON COMPUTERS*, 52(07):933–942, July 2003.

[3] G. Bollella, B. Delsart, R. Guider, C. Lizzi, and F. Parain. Mackinac: Making hotspot(tm) real-time. In *Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'05)*, pages 45–54, 2005.

[4] G. Bollella and J. Gosling. *The Real-Time Specification for Java*, volume 33. Addison-Wesley Publishing, 2000.

[5] G. C. Buttazzo and J. Stankovic. Red: A robust earliest deadline scheduling algorithm. In *Proceedings of Third International Workshop on Responsive Computing Systems*, 1993.

[6] A. Corsaro and D. C. Schmidt. The design and performance of the jrate real-time java implementation. In *On the Move to Meaningful Internet Systems, 2002 - DOA/CoopIS/ODBASE 2002 Confederated International Conferences DOA, CoopIS and ODBASE 2002*, pages 900–921, London, UK, 2002. Springer-Verlag.

[7] A. Corsaro and D. C. Schmidt. Evaluating real-time java features and performance for real-time embedded systems. In *RTAS '02: Proceedings of the Eighth*

IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'02), page 90, Washington, DC, USA, 2002. IEEE Computer Society.

[8] C. A. Francu. Real-time scheduling for java. Master's thesis, MIT Dept. of Electrical Engineering and Computer Science, 2002.

[9] G. Koren and D. Shasha. Rt-0138 - d-over : an optimal on-line scheduling algorithm for overloaded real-time systems. Technical report, INRIA, february 1992.

[10] J. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadline. In *11th IEEE Real-Time System Symposium*, pages 201–209, December 1990.

[11] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real time environment. *Journal of the Association for Computing Machinery*, 20(1):46–61, January 1973.

[12] C. D. Locke. *Best-effort decision-making for real-time scheduling*. PhD thesis, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, May 1986.

[13] Sun microsystems. *The Real-Time Java Platform, A Technical White Paper*, June 2004.