The Design and Implementation of Real-time Event-based Applications with RTSJ

Damien Masson and Serge Midonnet
Université de Marne la Vallée,
Institut Gaspard-Monge, Laboratoire d'informatique, UMR CNRS 8049,
77454 Marne la Vallée Cedex 2, France.
{damien.masson,serge.midonnet}@univ-mlv.fr

Abstract

This paper presents a framework to design real-time event-based applications using Java. The Real-Time Specification for Java (RTSJ) is well designed for hard periodic real-time systems. Though it also proposes classes to model asynchronous events and deal with sporadic or aperiodic tasks, it remains insufficient. The literature proposes the use of periodic servers called task servers to handle non-periodic traffics in real-time systems. Unfortunately, there is no support for task servers in RTSJ. In order to fix this lack, we propose an RTSJ extension model. To validate our design, we adapt and implement two policies: the polling server and the deferrable server policies. To show how efficient these policies are, we compare implementation results and results obtained with a discrete-event-based simulator.

1 Introduction

In order to bring to the real-time community the advantages of the Java language, a Java Specification Request (JSR-01) was proposed and accepted by the Java Community Process (JCP) in 1999. This lead to the Real-Time Specification for Java (RTSJ), which version 1.0 was released in November 2001 and first implementation in January 2002. Today, the specification is still evolving and commercial implementations have come out (Aïcas JamaïcaVM or more recently Sun Mackinac).

If this specification is well designed for periodic real-time systems, the support for non-periodic servicing with temporal constraints is not fully satisfying. In order to assign time and resource constraints to a group of threads, the RTSJ proposes the use of "Processing Group Parameters" (PGP). But as pointed in [1], no guidelines are given on how to use them, and there is no appropriate schedulability tests to analyse them. Furthermore, the cost enforcement is an optional behaviour for an RTSJ compliant virtual machine, and without this feature, PGP are useless.

That is why we propose in this paper a framework to design real-time event-based applications using RTSJ.

We present in Section 2 the aperiodic servers mechanisms and we detail the Polling Server and the Deferrable Server policies. Section 3 describes our framework to allow the implementation of these servers with RTSJ. Then we explain our implementation of the Polling Server and the Deferrable Server policies using this framework in Section 4. In order to evaluate the efficiency of our implementations, we developed a simulator that we present in Section 5. We give the results of these executions and simulations in Section 6, and discuss of future improvements in Section 7.

2 Tasks servers: presentation

During years, the assumption was made that real-time systems have to be only composed of periodic-tasks sets. This assumption comes from the feasibility analysis theories. How can we compute response times of tasks which we cannot predict the arrival of ? This is a too strong restriction since many of the real world phenomena are event-based.

One solution is to set up a mechanism which allows non-periodic traffic to be served and analysed without changing the feasibility conditions of a periodic task.

The easiest way to achieve this is to schedule all non-periodic tasks at a lower priority (we assume that the tasks are scheduled using a preemptive fixed priority policy). If it is very simple to implement, it does not offer satisfying response times for non-periodic tasks, especially if the periodic traffic is important. That is why periodic task servers are introduced by Lehoczky et al. in [8].

1-4244-0910-1/07/$20.00

A periodic task server is a periodic task, for which classical response time determination and admission control methods are applicable (with or without modifications). This particular task is in charge of servicing the non-periodic traffic with a limited capacity.

Several types of task server can be found in the literature. They differ by the way the capacity is managed. We can cite the *Polling Server* policy (*PS*), the *Deferrable Server* policy (*DS*), the *Priority Exchange* policy (PE) first described by Lehoczky et al. in [8] and developed in [11, 5, 9], the *Sporadic Server* policy (SS) presented in [10] and the *Slack Stealing* techniques introduced in [7].

Depending on the server policy, worst-case response time for the aperiodic tasks can or cannot be computed on-line when they occur. We have to separate periodic-tasks feasibility analysis - an offline process which can give guarantees on periodic-tasks (and servers) execution - and aperiodic-tasks feasibility analysis - an on-line process which can give guarantees at run-time on the aperiodic tasks execution.

2.1 Polling Server Principles

The *PS* is activated every period with its full capacity. If there are aperiodic tasks pending, it serves them with respect to its capacity limits and then loses its remaining capacity until its next activation.

Its most significant advantage is that it can be included in the feasibility analysis like any periodic task.

Assuming the server is the highest-priority task in the system, a feasibility test for the aperiodic tasks can be performed on-line. If the server is not the highest priority task, the complexity of the analysis becomes too high to be performed on-line, and since it cannot be performed off-line because of the unpredictability of the arrival model of aperiodics, it cannot be performed at all.

2.2 Deferrable Server Principles

The *DS* is activated as soon as an aperiodic event occurs (if it has enough capacity). It recovers its capacity every period.

The *DS* algorithm offers better average response-times than the *PS*, but since it can be activated with a delay, the feasibility analysis for the periodic tasks must be modified, as described in [11, 5].

As for the *PS* algorithm, complexity of the feasibility analysis for the aperiodics requires that the server runs at the highest-priority level in the system to guarantee the response times of aperiodics.

An exhaustive state of the art on this topic can be found in [2, chapter 5].

3 Tasks servers with Java: design

The RTSJ does not support any particular task server policy. It proposes two classes `AsyncEvent` and `AsyncEventHandler` to model respectively an asynchronous event and its handler.

The only way to include the handler in the feasibility process is to treat it as an independent task, and that implies to know at least its worst-case occurring frequency.

The RTSJ also provides the so called "Processing Group Parameters" (PGP), which allow programmer to assign resources to a group of tasks. A PGP object is like a `ReleaseParameters` which is shared between several tasks. More specifically, PGP has a `cost` field which defines a time budget for its associated task set. This budget is replenished periodically, since PGP has also a field `period`.

This mechanism provides a way to set up a task server at a logical level. Unfortunately it does not take into account any server policy. Moreover, as pointed in [1], it is far too much permissive and it does not provide appropriate schedulability analysis techniques.

Finally, since cost enforcement is an optional feature for an RTSJ-compliant virtual Java machine, PGP can have no effect at all. This is the case with the Timesys Reference Implementation of the specification (RI).

This is why we propose an RTSJ extension we can use to design and implement event based applications using task servers.

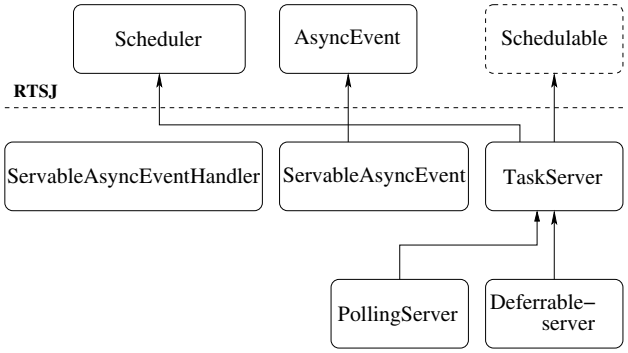

Figure 1. Classes to implement the server policies

Our framework (Task Server Framework) is composed of six new classes:

- `ServableAsyncEvent` (SAE)
  This `AsyncEvent` (AE) subclass represents a servable event. Like a normal AE, a SAE can be bound

to one or several standard handlers (i.e. `AsyncEventHandler`) using the `addHandler(AsyncEventHandler handler)` method. We overload it with the method `addHandler(ServableAsyncEventHandler handler)` and we redefine the method `fire()`.

- `ServableAsyncEventHandler` (SAEH)
  This class does not extend `AsyncEventHandler` (AEH), nor implement `Schedulable`. It embodies the code which can be associated with an SAE. It can be bound with one ore many SAE but associated with a unique `TaskServer`, and when one of the event it is bound with is released, it is added to the pending-events list of this server.

- `TaskServer`
  This abstract class represents a task server. It implements `Schedulable` and extends `Scheduler`. It is a schedulable object since it is in fact a periodic real-time thread and it is a scheduler since it has to schedule SAEHs. It has a method `servableEventReleased()` which takes an AEH. This method is called by the AE `fire()` method.

- `PollingTaskServer`
  A subclass of `TaskServer` which implements the *PS* policy.

- `DeferrableTaskServer`
  A subclass of `TaskServer` which implements the *DS* policy.

- `TaskServerParameters`
  A subclass of `ReleaseParameters` to construct a `TaskServer`.

Figure 1 shows dependencies between classes in the Task Server Framework and standard RTSJ classes.

To summarize the mechanism, when an SAE is fired, the `servableEventReleased()` methods of the bound servers are called for each of its SAEHs. This allows developers to write different behaviours for different task server policies: the handlers can be scheduled in a FIFO order, or any other desired order, depending on the implemented policy.

This design also allows programmers to use the `addTofeasibility()` methods on a `TaskServer` since it implements `Schedulable`. This approach is compliant with the RTSJ general design, but this is not sufficient. In order to provide a consistent design for the response times analysis, each schedulable object should have a `getInterference()` method, which would be called by the `Scheduler` feasibility methods. For example, it is still possible to compute response times in a system with a `Defferable Server`, but the algorithm has to be modified, and this is not possible in the centralysed RTSJ current approach. Anyway, this is not the matter of this paper.

4 Tasks servers with Java: Polling Server and Deferrable Server policies

The main issues during the conception of a task server are to guarantee the respect of its capacity and to ensure the capacity enforcement behaviour.

For the first problem, we used the `Timed` class. This class allows us to execute the `run()` method of an `Interruptible` object for a given maximum amount of time. If this maximum is reached before the `run()` method completes, the `AsynchrouslyInterruptedException` is raised and the `interruptAction()` method is called.

To control the server capacity, we make the assumption that the server cannot be preempted. Then, we just have to measure the time passed in the run method of the interruptible and decrease the remaining capacity accordingly.

Moreover, we are not able to resume a generic thread (even if it is explicitly interruptible), this is why a handler cannot be executed on multiple instances of the server. The system designer is in charge to split the treatment of each aperiodic event on multiple handlers, each one with a cost less than or equal to the server capacity (in fact less than because of the server's overhead).

So, our implementations of task servers policies are limited with the following constraints:

- the `worst case execution time (wcet)` of a non-periodic event treatment has to be less or equal to the server capacity;

- the server has to be the highest-priority task in the system.

4.1 Polling Servicing

Our class `PollingTaskServer` encapsulate a `RealtimeThread` with `PeriodicParameters`. The `run()` method of the server is delegated to this periodic real-time thread. When an asynchronous servable event is fired, its handler is added in a FIFO list. At each periodic activation, a method `chooseNextEvent()` is called. This method returns an `SAEH` or `null` if the server is not able to serve any pending event with its remaining capacity. While the chosen event is not `null`, it is executed (with the method `doInterruptible()` of `Timed`), the capacity is decreased and the `chooseNextEvent()` method is called again.

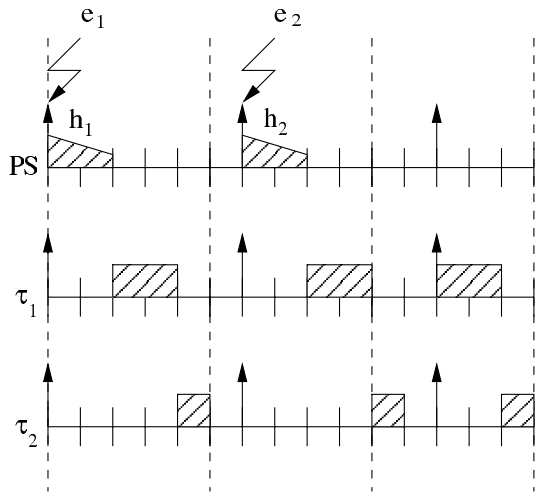

Figure 2. Scenario 1

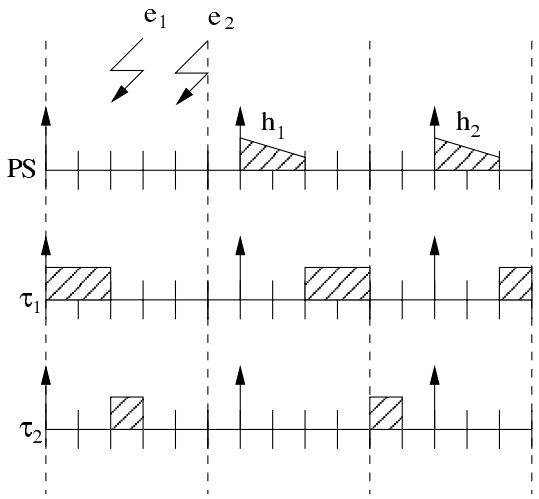

Figure 3. Scenario 2

This method return the first handler in the list which has a cost lower than the remaining capacity. This implies that if there is two handlers in the list, if the first - corresponding to the event released first - has a cost greater than the remaining capacity and if the second has a cost lesser than the remaining capacity, the event released last is served first.

Examples

	Priority	Cost / capacity	Period
PS	High	3	6
τ_1	Medium	2	6
τ_2	Low	1	6
h_1		2	
h_2		2	

Table 1. Tasks' Properties

In the following examples, the task set is composed of two real-time periodic tasks, τ_1 and τ_2, and one polling server PS running at the highest priority. These three periodic threads are synchronously started. There are two SAEHs, h_1 and h_2, respectively bound to two SAEs e_1 and e_2. Table 1 shows the properties of these tasks.

Scenario 1

Look at Figure 2, e_1 and e_2 are fired respectively at time 0 and 6. Since the server has its entire capacity at these two instants, h_1 and h_2 are immediately processed by the server.

Scenario 2

Look at Figure 3, e_1 and e_2 are fired respectively at time 2 and 4. We can see that h_2 does not begin its execution at time 8 because the remaining capacity of the server is 1, which is less than the cost of h_2.

With the real *PS* policy, h_2 should begin its execution at time 8, suspend it at time 9 and resume it at time 12.

Scenario 3

Without changing the code of h_2, we declare it with a cost of 1. Then, we fire e_1 and e_2 respectively at time 2 and 4. Figure 4 shows that h_2 begins its execution at time 8 because its cost parameter is set to 1, that is the remaining capacity, and is interrupted at time 9 because the server has consumed all its capacity and because h_2 has not finished.

With the real policy, h_2 should resume its execution at time 12, but it is not possible with Java.

4.2 Deferrable Server policy

Unlike the *PS*, the *DS* can serve an aperiodic task at any time as it has enough capacity. So the `run()` method can no longer be delegated to a periodic real-time thread. Instead, it is delegated to an AEH bound to a specific AE we call `wakeUp`. Each time an aperiodic event occurs, if the server is not already running, this event is fired. Moreover, we add a periodic timer which fire `wakeUp` if the server is not already running.

The second difference between the *PS* and the *DS* is that an event can begin its execution at the end of a server instance (for the *PS*, we made the assumption that the capacity is lesser than the period). So if the remaining capacity of

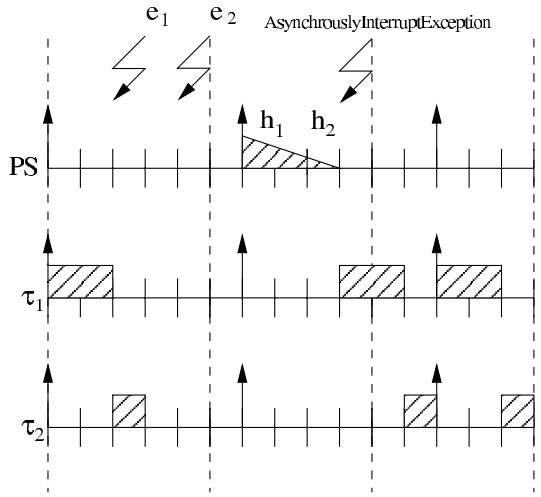

Figure 4. Scenario 3

the server is 1 when it has to serve an event with a cost of 2, if the next refill of the capacity is in a time lesser than 1, the event can be served. The `chooseNextEvent()` method now compare the current date with the next period: if the current date plus the chosen event cost is bigger than the next period of the server, the time budget associated with the event is equal to the remaining capacity plus the total capacity of the server.

Finally, as for the *PS*, it has to be noted that the implemented policy has not the exact behaviour of the real *Deferrable Server* described in literature.

5 RTSS: a real-time systems simulator

We develop a real-time event-based system simulator. This is a Java program which can simulate the execution of a real-time system and display a temporal diagram of the simulated execution. For now, three scheduling policies are implemented: Preemptive Fixed Priority, EDF and D-OVER.

This tool is distributed under the General Public Licence GNU (GPL), and can be found on the following web page: `http://igm.univ-mlv.fr/~masson/RTSS`

In order to compare with our task servers executions, we add the polling and the deferrable server policies to RTSS. The simulated policies are the ones described in literature: this is not a simulation of our implementations. Moreover, it does not take into account the servers overhead, nor the execution overhead.

6 Results

In order to validate our design, we implement the *PS* and the *DS* policy in Java and in an event-based simulator. We randomly generate task systems and we compare the simulations of these systems with their executions. The executions was performed using the reference implementation (RI) of the RTSJ on a 2 GHz processor INTEL Pentium 4 machine with 500 MB of memory and a rtlinux-free real-time kernel.

6.1 Real-Time Systems Generation

We develop a package `fr.umlv.randomGenerator` with a class `randomSystemGenerator` which can take the following parameters:

- **taskDensity**, the average number of aperiodic events per server period,
- **averageCost**, the average cost of aperiodic events,
- **stdDeviation**, the standard deviation of the aperiodic-events' costs
- **serverCapacity**, the server capacity,
- **serverPeriod**, the server period,
- **nbGeneration**, the desired number of generated system,
- **seed**, the random seed, in order to generate the same systems on multiple platforms.

We generate six sets of ten systems. The first set is defined by the tuple $(1, 3, 0, 4, 6, 10, 1983)$. This tuple permits to generate ten systems where the average number of aperiodic event per period is one, the costs of the events are $three$ time units (tu) (the average costs are $three$ and the standard deviations are $zero$), the servers capacities are $four\ tu$ and their periods are $six\ tu$.

In order to test the scalability of the system, we generate two other sets with the same parameters, except for the average number of aperiodics per instance: we generate one set with an average number of two and one with an average number of $three$.

Then we re-generate the same three sets but this time with standard deviations on the aperiodic events costs of two.

So we simulate the *PS* and the *DS* algorithms on these sixty systems and we execute them with our modified *PS* and *DS* implementation. We limit our simulations and executions on ten server periods.

	$(1,0)$	$(2,0)$	$(3,0)$
AART	8.86	17.52	23.76
AIR	0.00	0.00	0.00
ASR	0.89	0.63	0.43
	$(1,2)$	$(2,2)$	$(3,2)$
AART	10.24	20.58	25.50
AIR	0.00	0.00	0.00
ASR	0.85	0.50	0.35

Table 2. Measures on Polling Server simulations

	$(1,0)$	$(2,0)$	$(3,0)$
AART	12.24	20.80	25.05
AIR	0.01	0.01	0.00
ASR	0.75	0.44	0.30
	$(1,2)$	$(2,2)$	$(3,2)$
AART	6.55	7.15	12.54
AIR	0.17	0.24	0.29
ASR	0.48	0.34	0.30

Table 3. Measures on Polling Server executions

We measure the average response time of aperiodics, the interrupted-aperiodics ratio and the served-aperiodics ratio for each execution and simulation. Then we compute for each set the average of the average-response-times ($AART$), the average of the interrupted-aperiodics ratios (AIR) and the average of the served-aperiodics ratios (ASR).

The $AART$ give us a qualitative metric: the shorter the response times are, the more efficient the policy is. The AIR permit to estimate the overhead of the task server, since an event can be interrupted only if the server has theoretically enough resources to serve the event, but not enough in practice. Moreover, they can be used to adjust the acceptability threshold on the aperiodics' costs. Finally, the ASR are usefull to estimate the efficiency lost between our implementations and the theoritical algorithms of the *DS/PS* policies.

6.2 Polling Server results

6.2.1 Simulations

The measures on our *PS* simulations can be found on table 2. We can observe that the response times are a little greater when the event costs are not homogeneous. This is partly due to a bad-design issues on our costs generations: if a cost lower than 0.1ms is generated, we set it to 0.1ms. So the average cost has no longer the correct value. Even without this drawback, it can be explained by the fact that the highest cost a task has, the highest are the chances that the the server serve it in more than one instance, increasing its response time because of the server idle times.

6.2.2 Executions

Table 3 presents our Measures on the executions of our *PS* implementation. We have first to comment the average served ratios. They are lesser than the simulation ones. This is due to our not-resumable thread limitation: even if the server still has capacity, it has to delay the execution of task with a cost greater than its remaining capacity. This impact is reduced in the case of non-homogeneous task sets. Indeed, our server is able to execute a task released later if its cost is lesser than the capacity. For example, if the event queue contains two tasks τ_1 and τ_2, with $c_1 = 3$ and $c_2 = 1$, if the remaining capacity of the server is 2, then τ_2 can be executed instantaneously, even if it has been released after τ_1.

This server optimisation has an another consequence: the response times of events with low cost are improved. In the same time, there is more unserved task during the execution than during the simulation. In addition of the events which cannot be scheduled during the first ten periods of the server, there is the interrupted tasks, i.e. the ones which had overrun their costs due to the server's overhead. These interrupted tasks are mostly the ones with greater costs. These two facts lead to a far better average response time of served events in the execution than in the simulation. The lesser the costs are, the better the response times will be, as the chances to be not interrupted.

It has to be noted that the average interrupt ratio is very low for the homogeneous task sets. This is due to a simple fact: the server can only serve one event per period, since its capacity is 4 and the costs of the event are all 3. So each task has an additional time budget of $1s$ before being interrupted.

6.3 Deferrable Server

6.3.1 Simulations

Our simulations measures on the *DS* are presented on table 4. We can make the same observation than for the *PS*: the response times are a bit greater with the non homogeneous task set than with the homogeneous ones. Reasons are the same as in the previous case, increased because the deferrable server has a served ratio more important than the polling. This is due to its ability to serve each event as soon as it is released: the average response times are better than the polling ones and the events released during its tenth period can be served if there is no other pending events.

	$(1,0)$	$(2,0)$	$(3,0)$
AART	5.30	13.44	19.83
AIR	0.00	0.00	0.00
ASR	0.94	0.67	0.46
	$(1,2)$	$(2,2)$	$(3,2)$
AART	6.36	17.40	21.71
AIR	0.00	0.00	0.00
ASR	0.94	0.56	0.38

Table 4. Measures on Deferrable Server simulations

	$(1,0)$	$(2,0)$	$(3,0)$
AART	6.90	14.55	20.58
AIR	0.00	0.00	0.00
ASR	0.84	0.56	0.39
	$(1,2)$	$(2,2)$	$(3,2)$
AART	8.02	13.47	16.91
AIR	0.14	0.26	0.27
ASR	0.66	0.43	0.30

Table 5. Measures on Deferrable Server executions

6.3.2 Executions

Finally, the measures on the executions of our *DS* implementation can be found on table 5. Due to the ability of the server to serve the lower cost events in advance if their capacity is reduced and due to the lower served ratio, the response times of the execution are lower than the ones of the simulation for the non homogeneous task sets. We can note that the served ratios are very close to the simulations ones, that validates our implementations of task servers. We have to keep in mind that the simulations does not take into account the execution overhead.

7 Future Works and Improvements

The performance of our implementations can be improved. First, we have to reduce the average interrupted-aperiodics ratio (AIR). An interruption can have two different reasons: the task overruns its worst case execution time (WCET) - we cannot do anything - or the remaining capacity is too close to the cost of the event. Indeed, even if the server has to be the highest priority task in the system, there is also more highest priority tasks: the timers charged to fire the asynchronous events. We could decide that these timers have lower priority, but, in the case of the *PS*, this means that their executions will also be delayed until the next server period. Moreover, we could no longer measure the response times of the events. That is why we had not done it.

We can avoid some interruptions in delaying the execution of events handlers with a cost too close of the remaining capacity.

The over point we want to address is the on-line computation of event response times. Since the servers have to execute at the highest priority, a response time computation can reasonably be performed on-line at the arrival time of the event.

With the *PS* standard algorithm, assuming that the tasks are served in ascending deadline order and that the *PS* is running at the highest priority, the response time R_a of a task J_a - which is released at time r_a - can be computed on-line (at time r_a) with the following equations:

$$R_a = \begin{cases} t + C_{ape}(t, d_k i) - r_a & \text{if } C_{ape}(t, d_k) \leq c_s(t) \\ (F_k + G_k)T_s + R_k - r_a & \text{else.} \end{cases} \quad (1)$$

$$F_k = \left\lfloor \frac{C_{ape}(t, d_k) - c_s(t)}{C_s} \right\rfloor \quad (2)$$

$$G_k = \left\lceil \frac{t}{T_s} \right\rceil \quad (3)$$

$$R_k = C_{ape}(t, d_k) - c_s(t) - F_k C_s \quad (4)$$

where $C_{ape}(t, d_k)$ is the sum of the costs of the tasks with a deadline smaller than J_k, F_k the number of server instances needed to serve $C_{ape}(t, d_k)$, G_k the instance which begins to serve $C_{ape}(t, d_k)$ and R_k the time needed in the last instance to finish to serve $C_{ape}(t, d_k)$.

But our implementation suffers of some limitations: since a Java thread, even real-time, is not resumable, our *PS* only begins to serve a task if it has enough capacity to finish to serve it.

Taking into account these limitations, we can formulate the response time R_a of an aperiodic event J_a released at time r_a as:

$$R_a = (I_a T_s + C_{pa} + C_a) - r_a \quad (5)$$

where I_a is the instance of the server where J_a handler will be execute in, C_{pa} the cumulative cost of the previous handlers scheduled in the same instance and C_a the cost of J_a.

To easily compute J_a and C_{pa}, we propose a minor modification on our task servers implementations. Instead of put the events handlers in a simple FIFO list, we can set up a structure with a list of lists of handlers. Each list only contains a number of handlers which can be served in one single instance of the server. In addition, we can maintain another list of `RelativeTimes` which represents the cumulative costs of each handlers list. Now, I_a can be compute with the position in the list of lists of handlers where J_a has been

added and C_{pa} is the corresponding cost in the list of costs. Of course, this proposition will increase the time requested to register the release of a servable asynchronous event, but permits to compute in a constant time the response time of the event, and possibly to cancel its execution.

8 Related Works

The RTSJ is still subject to a lot of research, on the specification itself (JSR-282) as well as on specific extensions like the Distributed Real-time Specification (JSR-50) and the Safety Critical Java Technology (JSR-302). We can cite a recent publication of the JSR 282 in which the addition of several methods in the `RealtimeThread` class is suggested [6]. Other authors propose the use of model checking technics in order to highlight some issues in the RTSJ cost monitoring and enforcement model [3] and to analyse the behaviour for non-periodic real-time threads in the RTSJ [4]. In this paper, we have described the limitations of the RTSJ in regard to the support for aperiodic tasks.

9 Conclusions

The RTSJ does not support any aperiodic-task server mechanism. In this paper, we have proposed a framework to fill in this lack. In order to illustrate the use of this framework, we have adapted two task server policies: the Polling Server and the Deferrable Server ones. To validate our implementations, we also have implemented a simulator and a real-time system generator. We have compared the executions of 6 sets of 10 real-time systems with their simulations. We have observed better response times in our executions with served ratios very close to the ones from the simulations. There is still an important interrupted task ratio, but we have presented its causes and proposed a solution which we still have to test. Moreover, the simulations do not take into account the server overhead nor the costs of the events' release.

Finally, we have showed that we can easily implement a feasibility test at run-time for the aperiodic events, with a constant complexity. The implementation of this test, its integration in the RTSJ design and its confrontation with the response-times measured during the executions will make the object of our future works on this topic.

References

[1] A. Burns and A. J. Wellings. Processing group parameters in the real-time specification for java. In *On the Move to Meaningfull Internet Systems 2003: Workshop on Java Technologies for Real-Time and Embedded Systems*, volume LNCS 2889, pages 360–370. Springer, 2003.

[2] G. C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms And Applications*, volume 23 of *Real-Time Systems Series*. Springer Verlag, second edition, October 2004.

[3] O. M. dos Santos and A. Wellings. Cost monitoring and enforcement in the real-time specification for java - a formal evaluation. In *RTSS '05: Proceedings of the 26th IEEE International Real-Time Systems Symposium*, pages 177–186, Washington, DC, USA, 2005. IEEE Computer Society.

[4] O. M. dos Santos and A. Wellings. Formal analysis of aperiodic and sporadic real-time threads in the rtsj. In *JTRES '06: Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems*, pages 10–19, Paris, France, 2006. ACM Press.

[5] T. M. Ghazalie and T. P. Baker. Aperiodic servers in a deadline scheduling environment. *Real-Time Syst.*, 9(1):31–67, 1995.

[6] JSR-282 Expert Group. SI 1.4: Supporting Sporadic and Aperiodic Releases in Real-Time Threads. `http://jcp.org/en/jsr/detail?id=282`, 2006.

[7] J. P. Lehoczky and S. Ramos-Thuel. An optimal algorithm for scheduling soft-aperiodic tasks fixed priority preemptive systems. In *proceedings of the 13th IEEE Real-Time Systems Symposium*, pages 110–123, Phoenix, Arizona, December 1992.

[8] J. P. Lehoczky, L. Sha, and J. K. Strosnider. Enhanced aperiodic responsiveness in hard real-time environments. In *IEEE Real-Time Systems Symposium*, pages 110–123, San jose, California, December 1987. IEEE Computer Society.

[9] B. Sprunt, J. P. Lehoczky, and L. Sha. Exploiting unused periodic time for aperiodic service using theextended priority exchange algorithm. In *Real-Time Systems Symposium, 1988., Proceedings.*, number 0-8186-4894-5, pages 251–258, Huntsville, AL, USA, December 1988.

[10] B. Sprunt, L. Sha, and J. Lehoczky. Aperiodic task scheduling for hard real-time systems. *Real-Time Systems: The International Journal of Time-Critical Computing Systems*, 1:27–60, 1989.

[11] J. K. Strosnider, J. P. Lehoczky, and L. Sha. The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments. *IEEE Trans. Comput.*, 44(1):73–91, 1995.