RTSJ Extensions: Event Manager and Feasibility Analyzer

Damien Masson, Serge Midonnet

RTSJ Extensions: Event Manager and Feasibility Analyzer

Damien Masson and Serge Midonnet
Université Paris-Est
Laboratoire d'informatique Gaspard-Monge
UMR 8049 IGM-LabInfo
77454 Marne-la-Vallée Cedex 2, France
{damien.masson, serge.midonnet}@univ-paris-est.fr

ABSTRACT

We present in this paper our experience on the implementation with RTSJ of advanced algorithms to handle aperiodic traffic. We have adapted existing algorithms in order to take into account some constraints brought about by the use of Java language, and our aim which is to propose a portable mechanism. To circumscribe some difficulties, we had to use some programming ruses which could be better integrated into the specification. From these experiences resulted a set of modifications to the specification which we propose to submit to the community in this paper, in addition to a unified event manager framework.

1. INTRODUCTION

The aim of the Real-time Specification for Java is to design APIs and virtual machine specifications for the writing of real-time applications in Java language. An important aspect of real-time system programming is the feasibility analysis to ensure the respect of temporal constraints.

The case of hard real-time systems composed of periodic tasks was extensively elaborated for each of the many scheduling policies. The RTSJ is well designed to write such systems, with feasibility analysis methods integrated both in `Scheduler` abstract class and `Schedulable` interface.

In more realistic systems composed of hard real-time periodic tasks and soft real-time aperiodic events, three approaches are possible to ensure that the interference of aperiodic tasks on periodic ones is bounded: 1) scheduling the aperiodic tasks with a lower priority ; 2) bounding the minimal inter-arrival time of aperiodic events, and studying the worst case scenario where they arrive at this worst rate ; 3) delegating the service of non periodic events to a mechanism which can be integrated into the analysis.

There are classes in the RTSJ to model handlers associated to asynchronous events. These handlers are schedulable objects which can be set either with `AperiodicParameters` or with `SporadicParameters`. The latter enables the event to be integrated into the feasibility analysis process as a stand alone task, as if it were arriving at its maximal rate.

For the third approach, the `ProssessingGroupParameters` class is proposed. It enables several schedulable objects to share release parameters, such as a mutual CPU time periodic budget. Unfortunately, it will not support any aperiodic task server policy nor any other advanced mechanism to handle aperiodic events. Moreover, as pointed out in [1], this is far too permissive and does not provide appropriate schedulability analysis techniques.

Therefore, if the sporadic approach is not possible or too pessimistic, the only remaining solution with RTSJ is to schedule the task in the background.

In [7] we began to address the problem of event managing with an API proposition to write task servers. We continued this work and developed an approximate slack stealer compatible with the task servers model in [8].

In this paper, we propose to generalize our task server model into an event manager model. We want to set up a homogeneous event manager framework and propose modifications to the specification in order to better include this framework. What we propose exactly is this: to modify the RTSJ feasibility analysis approach and to add methods to the `Scheduler` class for monitoring tasks execution and inserting treatments before and after each schedulable object execution.

The remainder of the paper will be structured as follow: we shall discuss in Section 2 which programming level is suitable to write event handling mechanisms. We present a task server framework and a slack stealer for RTSJ respectively in Sections 3 and 4. We also discuss modifications on the specification in order to better integrate them. Some results are commented on in Section 5. We expose the limitations of the feasibility analysis model of the RTSJ and propose to extend it in Section 6. Finally we conclude in Section 7 where we recapitulate the extensions which we propose for the RTSJ.

2. PROGRAMMING EVENT HANDLING

The aim of a scheduler is to schedule tasks. It is responsible for handling the execution of pending tasks by following

JTRES'08, September 24-26, 2008, Santa Clara, California, USA

```
public class MyRealtimeThread extends RealtimeThread{
  ...
  public static boolean waitForNextPeriod(){
    computeAfterPeriodic();
    boolean returnValue =
      RealtimeThread.waitForNextPeriod();
    computeBeforePeriodic();
    return returnValue;
  }
  ...
}
```

Figure 1: `WaitForNextPeriod()` modifications

a specific policy. Thus event handling should be programed in the scheduler. Unfortunately, this is not always possible.

The specification contains an abstract class, `Scheduler`, the aim of which is to model the system scheduler. The scheduler class can be overridden and a multiplicity of implementations can be proposed. However, if we take a look at the implementations of this class in available RTJVMs, we find that scheduler implementations are provided by native methods, which use the underlying operating system scheduler. This means that the use of a scheduling policy which is not provided by the system your virtual machine is running on, is impossible. Besides, the RTSJ only imposes a preemptive fixed priority scheduler to each RTSJ-compliant JVM.

Thus if one wants to design a portable mechanism for event handling or another advanced algorithm to deal with specific tasks in a given system, one has to provide it in user land, above the underlying fixed priority scheduler.

Several questions need to be answered here: how and when can we control the system without being the scheduler ? How can we ensure the respect of timing constraints without using underlying system specific data ? In the following subsections, we will look at solutions to these problems.

2.1 Carrying Out Computations

In order to control the system, the exact implementation details of the scheduler must be known and the highest priority task must be used to avoid interference from other tasks. Unfortunately this interferes with the business tasks. The mechanism has to be integrated into the feasibility analysis process, and must be periodic, or have known activation behavior, so that its interference with other tasks can be easily confined.

The solution we propose consists of performing necessary computations just before the beginning of the periodic tasks, and just after their completion. When doing this, if the cost value of these computations is known, then this value can be added to the periodic tasks worst case execution cost, and then the usual feasibility analysis can be performed.

This solution is easily integrable with the RTSJ. Indeed, the periodic behavior is obtained with the specification by writing the code in a loop and by calling the special method `waitForNextPeriod()` at the end of this loop.

This method call is blocking and the thread is activated automatically by the virtual machine at its next activation. Thus we just have to extend `RealtimeThread` or other `Schedulable` object and override the `waitForNextPeriod()` method in the way shown in Figure 1. Then it only remains to write the two methods `computeBeforePeriodic()` and `computeAfterPeriodic()`.

Although this design may seem to be just a patch, it is extremely useful. For example, in [6] we used it to set up temporal fault detectors, and as we will see in Section 4 it enables computations to be carried out to estimate the available slack time in the system.

In a system composed only of periodic tasks, it can be used to find out the amount of time a task has really consumed. A task stops its execution only when it has completed its periodic activation or when a higher priority task begins its own execution. Thus if the elapsed time is measured between the last time an entry is made into one of the two proposed methods, the consumed CPU time for each task can be kept up-to-date. It supposes that the execution stack is kept in order to know which is the executing task between the two calls.

This behaviour is missing from the RTSJ and has been proposed by the JSR-282: *"7. Add a method to Schedulable and processing group parameters, that will return the elapsed CPU time for that schedulable object (if the implementation supports CPU time)".*

With our *patch* we can obtain the elapsed CPU time even if the implementation cannot support the method which provides that elapsed CPU time. A drawback is that the cost must be paid both in memory and CPU usage.

The proposed `MyRealtimeThread` class (Figure 1) requires that all the tasks in the system either extend or use it. If one uses both `MyRealtimeThread` and regular `Schedulable` objects, one can no longer deduce anything from the elapsed time since the last beginning or end of a `MyRealtimeThread` implemented task. Moreover, the patch becomes inefficient if `AsyncEventHandler`s are used to write periodic tasks, or if the tasks share resources, inducing priority inversions. So it is, in our opinion, a good idea to integrate this mechanism with the RTSJ. In order to do so, we propose to modify the `Scheduler` abstract class by adding abstract methods automatically called before tasks instances start and after their completion, for all schedulable objects. Appropriate methods can also be called for other context switches due to monitor control algorithms or tasks which suspend themselves.

Then we also propose to add a CPU time monitor to the `Scheduler` which uses this mechanism. This monitor can be turned off if it is not needed or if the cost enforcement is available on the targeted platform. If it is turned on, a method `RelativeTime getConsumnedCPUTime(Schedulable s)` should return the total amount of time the `Schedulable s` has consumed before its last activation. Another method `getTotalCPUTime(Schedulable s)` can return the total CPU time consumed by the schedulable. If priority inversion due to resource sharing has occurred, the blocking

time values can be incorporated or return separately.

As this mechanism has a non negligible cost, it has to be integrated into the feasibility analysis. In order to do so in a transparent way, a method `getContextSwitchCost()` should be added in the `Scheduler` class.

All the extension propositions we make are summed up in Section 7.

2.2 Ensuring Timing Constraints

Let us suppose here that an event management algorithm allows us to start an `AsyncEventHandler` h at the priority p, assuming that the worst case execution time of h is C_h.

We need to ensure that h does not exceed this worst case execution time, since the algorithm relies on this parameter to allow us to start h at the priority p.

Cost enforcement is an optional behavior in the specification, so we do not want to use it. The only remaining solution is to set up a timer which suspends the task if it has not completed its execution at the expiration. But the consequence is that we need to cancel the timer each time the task is preempted, and resume it each time the task is resumed.

We can do that with the patch proposed in Section 2.1, but the overhead of these timers' management would be linear with the number of pending tasks.

Such a cost can neither be bounded nor integrated in the feasibility analysis since the number of aperiodic tasks cannot be controlled. This induces a strong limitation in the user land approach to managing non periodic events: we have to schedule them at the highest available priority. When doing that, we ensure that we need only one control timer at the same time and this timer is set up once and for all.

The other limitation is that if a thread can be suspended through the `AsynchronouslyInterruptedException` mechanism, it is not possible to resume it. This means that we need to adapt existing algorithms available in real-time scheduling literature. Indeed, available resources (server remaining capacity or slack) can exist but may be lower than the worst case execution cost of any pending aperiodic events. This leads to situations where some resources available in the general context of real-time systems are not expendable because of the use of the RTSJ. To limit this side effect, we can take action with the queue policy in the event manager.

Moreover, for the same reason, a task with a very high cost can never be carried out. To avoid this drawback, we imagined a BS-duplication policy which consists of simultaneously scheduling each aperiodic task in background and enqueuing it in the event manager. The first replication which completes its execution interrupts the other.

3. TASK SERVERS IMPLEMENTATION

Task servers which were first introduced in 1987 [5], are special tasks in the system which are in charge of servicing

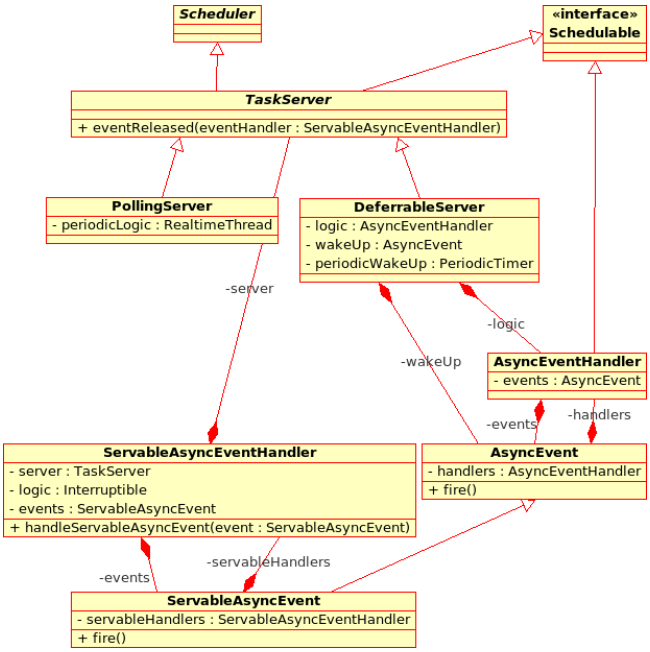

Figure 2: Classes to implement the server policies

aperiodic traffic. They are characterized by a capacity and a policy to recover it. The execution of the aperiodic traffic cannot interfere with the system more than a server using its full capacity. The main advantage is that these special tasks can be integrated into the feasibility analysis (with or without modifications to it).

Many task server algorithms have been proposed in the two previous decades [10, 9]. However, 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 consider it as an independent task, and that implies knowing its worst-case occurring frequency. Even if the maximum value of the frequency is known, it often represents a very pessimistic approximation.

The RTSJ also provides the class `ProcessingGroupParameters` (PGP), which allows programmers 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 tasks 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 out in [1], it is far too permissive and it does not provide appropriate schedulability analysis techniques. Finally, as cost enforcement is an optional feature for a RTSJ-compliant virtual Java machine, PGP may not be available. This is the case with the Timesys Reference Implementation of the specification (RI).

We proposed a set of classes to write task servers in [7].

This contribution is summed up by Figure 2. It is composed of six new classes: `ServableAsyncEvent` (SAE), `ServableAsyncEventHandler` (SAEH), `TaskServer`, `PollingServer` and `DeferrableServer`. The notion of "Servable" object is similar to the "Schedulable" one, except that if a `Schedulable` is executed by a scheduler, whereas a `Servable` is handled by a server. Thus, the `Schedulable` object is the server. A `ServableAsyncEvent` extends the regular `AsyncEvent`.

It models an asynchronous event where its handlers can be either asynchronous event handlers, i.e. regular `Schedulable` objects, or `ServableAsyncEventHandler`s: shells containing logic that task servers can execute.

Each `ServableAsyncEventHandler` has to be registered in one `TaskServer`. Then, when a `ServableAsyncEvent` is fired, each of its regular `AsyncEventHandler`s is set ready for execution. For each of its servable handlers the `servableEventReleased(SAEH h)` method of its associated server is invoked.

After this the server is notified and can, for example, enqueue the handler. 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. Finally `TaskServer` inherits from `Scheduler` in order to enable the use of the feasibility methods for aperiodic tasks inside the server.

3.1 Polling and Deferrable Server Modifications

As a demonstration of the efficiency of this design, we implement two well known service policies: the Polling Server and the Deferrable Server, taking into account the limitations described in Section 2 to modify these policies.

The servers have to run at the highest priority, which prohibits the use of several servers. As we cannot resume the threads, we start a handler only if the remaining capacity in the server is equal to or greater than its worst case execution time.

This leads to situations where the server still has capacity, and has tasks to execute, but remains inactive. In the case of the deferrable server, the loss in performance is limited, as the server has bandwidth conservation, but the polling server loses its remaining capacity when it becomes inactive.

To limit the loss of performances, we investigated a lot of queue policies. We tried a simple FIFO, a LIFO, scheduling first the task with the highest cost (HCF) and finally the one with the lower (LCF). The policy which perform better in all the cases is the LCF policy. Moreover, to ensure that all task can be scheduled, we set up a policy we called "BS duplication" as explained in Section 2.2. Figure 3 show results of simulations we conducted to estimate the loss of performances. Surprisingly, with the impact of the BS duplication, the policies performances are quiet similar. We even have the modified DS policy which performs better than the preemptive one in some cases.

From an implementation point of view, there is no specific

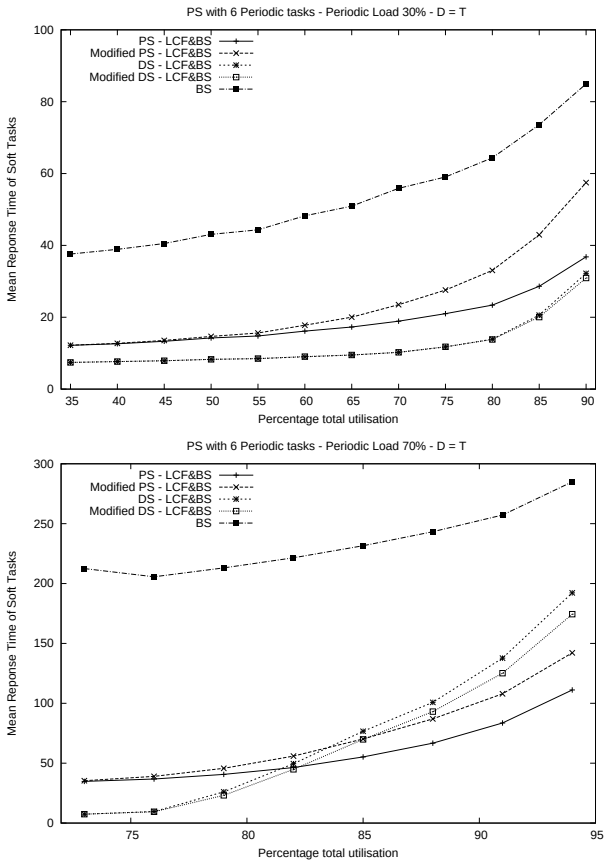

Figure 3: Regular server policies VS modified ones

problem with the Polling Server. We use a delegation to a `RealtimeThread` set up with `PeriodicParameters`. Writing the code for the deferrable server was a little bit more tricky, since it can be activated at any moment. So we use a delegation to an `AsyncEventHandler` associated to a special `AsyncEvent` "wakeUp". This event is fired each time a `ServableAsyncEventHandler` is added to the waiting queue of aperiodic requests (i.e. each time a `ServableAsyncEvent` is fired). This handler is also associated to a `PeriodicTimer` in order to manage the capacity.

Despite the limitations and adaptations to the policies, the performances are much better than just executing aperiodic service as a background task. However, the extensive evaluation of performances is not the purpose of this paper.

3.2 Integrating Servers and Feasibility Analysis

Integrating the Polling Server in the feasibility analysis process is straightforward. Indeed, it is just a regular periodic task in the worst case (when it uses its full capacity). The general overhead of the mechanism can be deducted from the server capacity. We use a `PeriodicParameters` object in which the field `cost` is used for the capacity.

The Deferrable Server induces more difficulties. In fact, the feasibility analysis has to be modified because in the worst case, this server does not interfere like a regular pe-

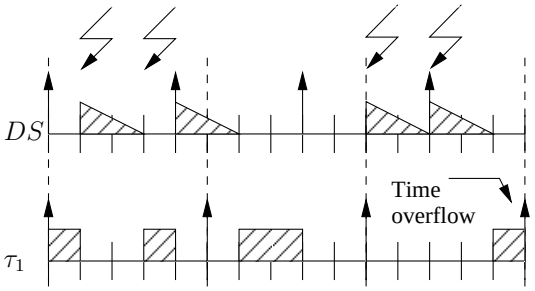

Figure 4: Double hit effect

riodic task on the lower priority tasks. This is due to its ability to conserve its capacity when there is no traffic to the server. The counter example of a feasible system with a Polling Server but not feasible with a Deferrable Server is well known as the *double hit* effect. It is represented in Figure 4.

A sufficient and necessary condition for the feasibility is available and described in [3]. However, with the current specification, the only way to change the feasibility analysis algorithm is to overrun the `Scheduler` class. From our point of view, this is not a coherent action if the scheduling policy is not modified, which is the case. We propose the creation of a new interface `FeasibilityAnalyzer`, which can be integrated into the scheduler as a field, and can be changed by a setter method call. This question will be further discussed in Section 6.

4. A SLACK STEALER FOR RTSJ

For the general problem of jointly scheduling hard real-time periodic tasks and soft real-time non periodic events, the best known solution for minimizing the soft tasks' response times whilst guaranteeing hard tasks deadlines is the use of a slack stealer, proposed in [4] and [2]. An approximate algorithm, DASS, is also presented in [2]. We proposed in [8] MASS, an algorithm to estimate the slack using data only updated when periodic tasks end or begin. This algorithm is more pessimistic than DASS but needs less computations, and only when tasks ends. The slack at the instant t is the total amount of time you can suspend all the tasks without inducing temporal faults (i.e. deadline misses).

We perform $\mathcal{O}(n)$ complexity operations each time a task ends, and this allows us to compute a lower limit on the available slack in constant time.

Implementing this algorithm with RTSJ is simple as it is designed to take into account RTSJ userland restrictions evoked in Section 2. Indeed, the evaluation relies on two pieces of data for each task kept up to date whenever a task begins and ends.

4.1 Estimating the Slack

We consider a process model of a mono processor system, Φ, made up of n periodic tasks, $\Pi = \{\tau_1, ..., \tau_n\}$ scheduled with fixed priorities. Each $\tau_i \in \Pi$ is a sequence of requests for execution characterized by the tuple $\tau_i = (C_i, D_i, T_i, Pi)$. Where C_i is the worst case execution time of the request ; D_i is its relative deadline; T_i its period and P_i its priority, 3 being the highest.

The system also has to serve an unbounded number p of aperiodic requests, $\Gamma = \{\sigma_1, ..., \sigma_p\}$. A request $\sigma_i \in \Gamma$ is characterized by a worst case execution time C_i.

Let $S_{i,t}^{max}$ denotes the slack at priority level i available at the instant t, i.e. the amount of time the processor will be idle for priority levels higher or equal to i between t and the next τ_i deadline. Then S_t^{max}, the available time at the highest priority at time t is the minimum over all the $S_{i,t}^{max}$. This quantity can increase only when a periodic task ends its periodic execution. So we propose to keep up-to-date for all task $S_{i,t}$, a lower bound on $S_{i,t}^{max}$. S_t is computed each time a periodic task ends, and is assumed to have decreased by the elapsed time otherwise. So the $S_{i,t}$ values only have to be correct (i.e. lesser than or equal to $S_{i,t}^{max}$) in such situations.

$S_{i,t}$ is described using two elements: firstly a lower bound on the maximum possible work at priority i regardless of lower priority processes, $w_{i,t}$, secondly the effective hard real-time work we have to process at the instant t, $c_i(t)$.

Equation 1 recapitulates the operations needed to obtain a lower bound on the available slack at time t. These operations have an $\mathcal{O}(n)$ time complexity.

$$\begin{aligned} \forall i, S_{i,t} &= w_{i,t} - c_i(t) \\ S_t &= \min_{i \setminus \tau_i \in \Pi} S_{i,t} \end{aligned} \tag{1}$$

Two pieces of data must be kept up-to-date for each periodic task: $w_{i,t}$ and $c_i(t)$.

$$c_j(t) = c_j(t') - \min(dt_c, dt_w) \tag{2}$$

$$\begin{aligned} \forall j < i, w_{j,t} &= w_{j,t'} - dt_w \\ w_{i,t} &= w_{i,t'} + T_i - I_i \\ \forall j > i, w_{j,t} &= w_{j,t'} + dt_w + C_i \\ c_i(t) &= C_i \end{aligned} \tag{3}$$

If we note t the current time, t' the time of the last update of a value, dt_c the elapsed time since the last update of a $c_k(x)$ value, dt_w the elapsed time since the last operation on a $w_k(x)$ value, τ_i the task which begins or ends and τ_j the task which is executing just before t (if any), then Equation (2) indicates the operation which has to be performed at the beginning of the task τ_i and Equation (3) indicates the operations which have to be performed at the end of the task τ_i. Equation 3 still holds even if we allow resources sharing and priority inversions (we just have to replace C_i by $C_i + B_i$), but the operation described by equation 2 has also to be performed when a critical section is entered.

At the end of a τ_i periodic execution, the work available :

- is increased by T_i minus the interference from higher priority tasks activated during its next instance for τ_i;

- is decreased by dt_w but increased by C_i for tasks with a lower priority;
- is decreased by dt_w for tasks with a higher priority.

The interference, which we shall call I_i in Equation (3), can take two different values depending on the instance. The most accurate value can be found in a constant time complexity operation.

The operations needed when a task completes all have a time complexity linear in the number of tasks. Their cost can be bounded to the worst case execution cost C_{ep}. Respectively, the operation needed when a periodic task begins has a constant time complexity and can be bounded to the worst case execution cost C_{bp}. These values can be integrated into the feasibility analysis process.

This integration is not harmonious with the current RTSJ. If we use the `MyRealtimeThread` class proposed in Figure 1, we have to increase the cost of each task by $C_{ep} + C_{bp}$. A more elegant way to proceed is to use additional methods in the `Scheduler` class to add the computation before and after all the `Schedulable` executions, and to add a new parameter to the `Scheduler` class which represents the context switch cost.

4.2 Using the Slack

A slack stealer can be viewed as a task server with a capacity which is always equal to the available slack. Thus when an aperiodic task arrives, the slack can be estimated in a constant time operation. After that, as with the task servers, multiple policies are possible.

We can add a new class `SlackStealer` to Figure 2. This class extends `TaskServer`. The slack stealer has to be activated aperiodically: when an aperiodic task is released while the queue is empty, and when the slack time is increased while the queue is not empty. So we can use the same mechanism we used for writing the code of the `DeferrableServer`: The logic of the `SlackStealerTaskServer` class can be delegated to an `AsyncEventHandler` associated to a special `AsyncEvent`.

However, the slack stealer approach is not really similar to that of the server. The server is similar to a reservation approach whilst the slack stealer uses the unused available resources. So we prefer to rename our class `TaskServer` in `EventManager`.

5. SOME RESULTS

Figure 5 exposes some simulation results with modified server and RTSJ compliant slack stealer algorithms. MASS designates the algorithm implementable with RTSJ using the slack approximation while ESS and DASS designate the same algorithm with respectively an exact knowledge of the available slack and a slack approximation obtained by DASS algorithm.

We compared a lot of queuing policies, and the best one consisted of first scheduling the task with the lowest cost. We named this policy LCF (Lower Cost First).

The use of the BS-duplication is noted "&BS".

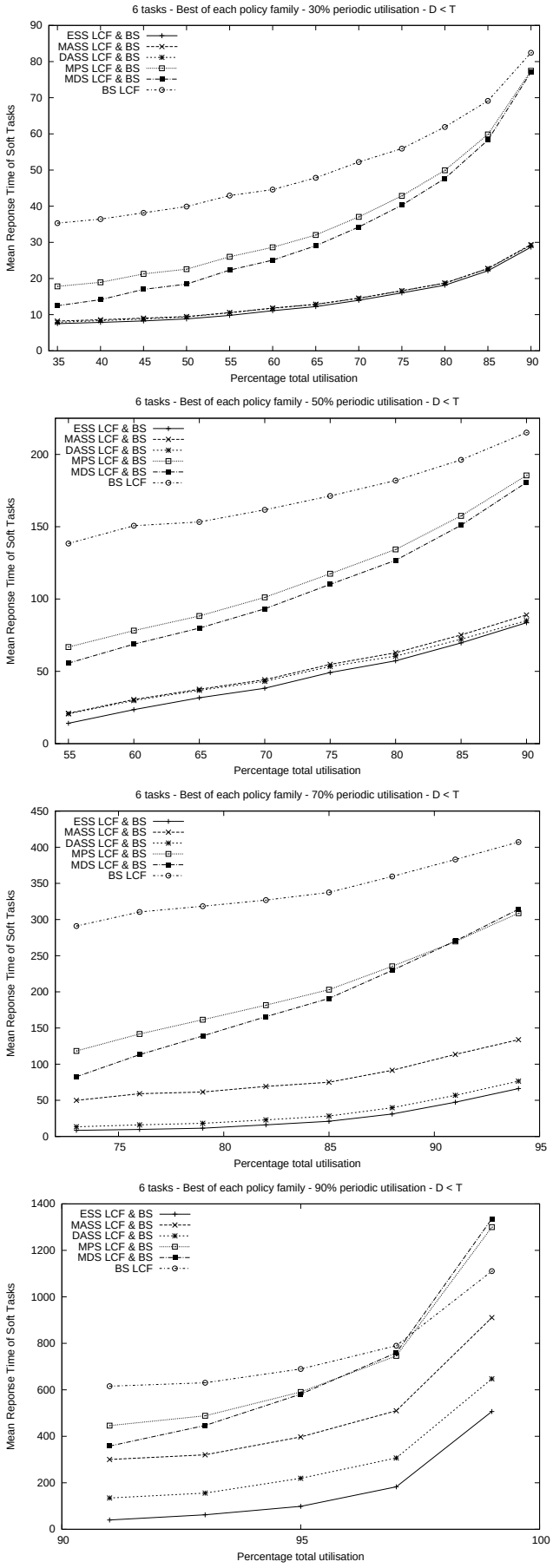

Figure 5: RTSJ compliant Slack Stealer and Task servers algorithms

We measured the mean response time of aperiodic tasks with different aperiodic and periodic loads.

First, we generated groups of ten periodic task sets with utilization levels of 30, 50, 70 and 90%. The results presented are averages over a group of ten task sets.

We conduct the experiments with a variable number of periodic tasks from 2 to 100. The periods are randomly generated with an exponential distribution in the range [40-2560] time units. Then the costs are randomly generated with an exponential distribution in the range [1-period] and deadlines with an exponential distribution in the range [cost-period]. Priorities are assigned assuming a Deadline Monotonic Policy.

Non feasible systems are rejected, the utilization is computed and systems with an utilization level differing by less than 1% from that required are kept.

Then, we generate groups of ten aperiodic tasks sets with a range of utilization levels (plotted on the X-axis in the following graphs). Costs are randomly generated with an exponential distribution in the range [1-16] and arrival times are generated with a uniform distribution in the range [1-100000]. Our simulations end when all soft tasks have been served.

The figure presents the best policy for each algorithm (BS, MPS, MDS, MASS, DASS and ESS) on systems with $D_i < T_i$ for all periodic tasks. Equivalent results are obtained on systems with $D_i = T_i$. For all load conditions, servers bring real improvement compared to BS. The DS offers better performances than the PS. MASS performs better than the DS, DASS better than MASS and ESS better than DASS. For systems with periodic loads of 30% and 50%, results obtained with MASS, DASS and ESS are quite similar. Considering the differences between these algorithms' time complexities (linear with very low constant, linear with a high constant and pseudo polynomial), this is a very satisfying result. However MASS performances degrade quickly than DASS ones when periodic load increases. Nevertheless MASS remains a really good implementable algorithm even for systems with a periodic load of 90%.

6. FEASIBILITY ANALYSIS DESIGN

The feasibility process in the RTSJ is not suitable for mixed task systems. Indeed, the problem is that the methods are part of the scheduler class and this has several disadvantages.

The choice of feasibility analysis (FA) algorithms depends on the type of application. A simple statistical condition can be suitable for a multimedia application. Here, a deadline miss can be acceptable, if the analysis ensures that after m periods, k instances of the task can be executed.

In other cases, a necessary but non sufficient condition on the feasibility can be acceptable. For example if a detection mechanism and a temporal fault gesture is set up at runtime, or if the worst case execution times are known to have been over-evaluated.

For other applications, a sufficient test is needed, a test which can be necessary or not.

However, the FA is highly dependent on the scheduling algorithm. So in our opinion, the feasibility analyzer has to be a separate object by itself, but must be integrated into the scheduler object as a parameter, which the developers can change according to the target application they are writing.

Of course, this is already possible with the current specification by overriding the `Scheduler` and changing the FA methods, but this is not really a coherent approach. Indeed, the FA does not have to affect the scheduling behavior, and changing the analysis algorithm should not mean changing the scheduler object since the task scheduled will remain the same.

So we propose the addition of an interface `FeasibilityAnalyzer` in the RTSJ. A field of `Scheduler` can be typed with this interface, and all the methods relative to the FA delegated to it.

Then it is possible to change the default FA on demand. This interface should have the methods `addToFeasibility(Schedulable s)` and `boolean isFeasible()`. We propose two other interfaces extending `FeasibilityAnalyzer`: `SufficientTest` and `NecessaryTest`. Then we can set up a fourth interface `SufficientAndNecessaryTest` which extends the others two.

In this way, if a class can use any FA policy, it can just type its field `FeasibilityAnalyzer`, but all the other degrees of precision can be enforced just by the correct choice of the type.

We can easily write the class `LoadCondition` as an implementation of `NecessaryTest`. This class must compute the system load (for fixed priority scheduled systems : $U = \sum C_i/T_i$).

For the fixed priority preemptive case, which is the only scheduling policy imposed for an RTSJ compliant JVM, we can add the abstract class `ResponseTimeAnalysis` as an implementation of the interface `SufficientAndNecessaryTest`, and finally the class `FixedPriorityRTAnalysis` as its subclass.

Possible methods for this abstract class are `RelativeTime computeLevelIBusyPeriod(int i)` which computes the i-level busy period and `computeResponseTime(int i, int q)` which computes the response time of the instance q of the task τ_i.

7. CONCLUSIONS

In this paper, we have shown how an aperiodic event traffic can be handled using the RTSJ implementation in a portable way. We have presented the necessary adaption of existing algorithms in order to take into account the constraints brought about by the use of the Java language. We have presented a set of modifications to the RTSJ Specification and a unified event management framework. This framework enables RTSJ programmers to write a simple and

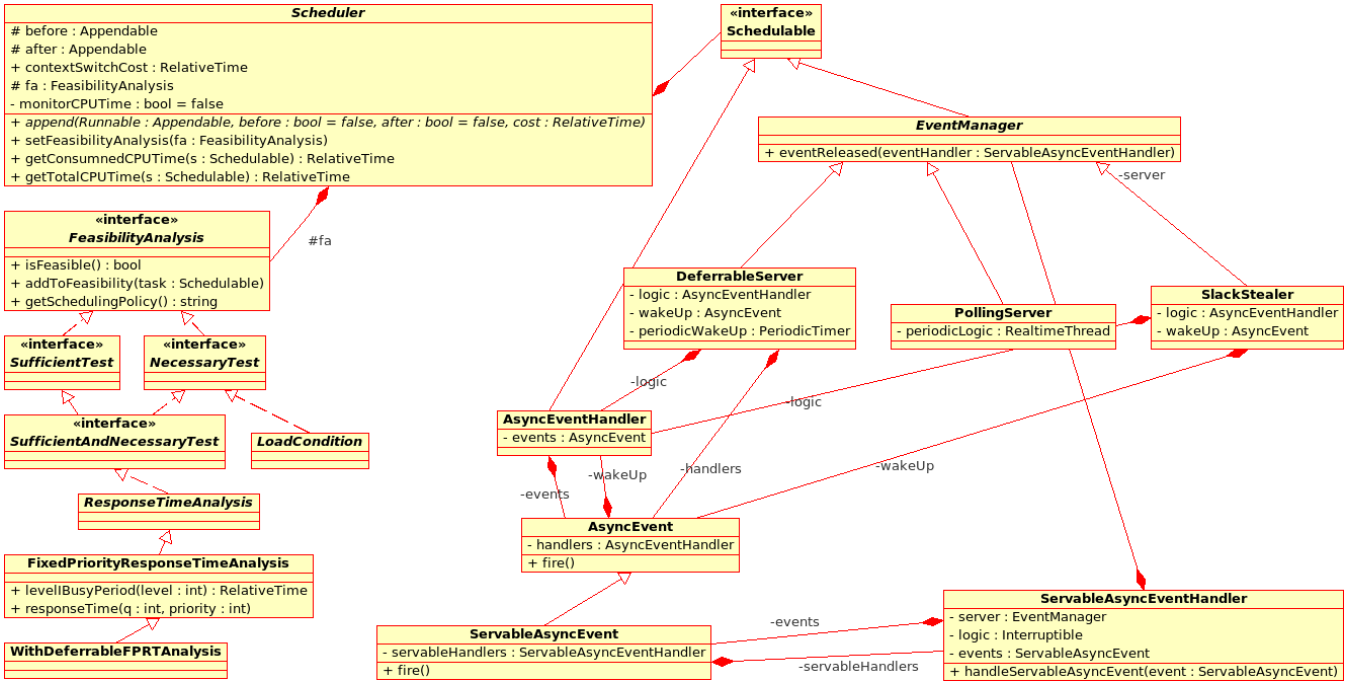

Figure 6: Extensions to the RTSJ we propose

portable application code.

We now recapitulate the modifications and additions which we consider to be relevant in order to improve the specification. Then, the RTSJ extension proposed is the following:

- add `void computeBefore(Runnable logic, RelativeTime cost)` and `void computeAfter(Runnable logic, RelativeTime cost)` methods in `Scheduler` abstract class. With these methods the scheduler automatically adds `logic` parameter before or after each instance of periodic or non periodic tasks handled. The parameter `cost` is the logic worst case execution time;
- add the boolean field `monitorCPUTime` and the two methods `RelativeTime getConsumedCPUTime(Schedulable s)` and `getTotalCPUTime(Schedulable s)` to the class `Scheduler`, in order to allow the CPU time consumption monitoring when the underlying operating system does not directly provide this feature;
- add a `contextSwitchCost` field typed `RelativeTime` and its getter method to the `Scheduler` abstract class, in order to integrate the context switch cost, the logic added by the previously proposed methods and the optional CPU monitoring mechanism to the feasibility analysis;
- add new classes `ServableAsyncEvent` and `ServableAsyncEventHandler`. The first extends the class `AsyncEvent` and models an event which can be associated both to regular schedulable handlers and special handlers associated to an event manager;
- add the new abstract class `EventManager`, and its implementations `PollingServer`, `DeferrableServer`, `SlackStealer`. We can provide the Java code for these three classes;
- add the interfaces `FeasibilityAnalyzer`, `SufficientTest`, `NecessaryTest` and `SufficientAndNecessaryTest`;
- add the abstract class `ResponseTimeAnalysis` and its subclass `FixedPriorityRTAnalysis`;
- integrate these feasibility relative classes and interfaces into the `Scheduler` abstract class as a field and with setter/getter methods. Delegate the behaviors of existing feasibility analysis methods to this field.

Figure 6 recapitulates these propositions in a UML diagram.

In future work, we have to investigate on the interactions between the memory parameters and the feasibility analyzer object. We also have to clarify the behavior of the proposed CPU time user land module when resource sharing is allowed.

Acknowledgments

We want to thank Sian Cronin and Aurore Sibois for their valuable advices and English corrections.

8. REFERENCES

[1] Alan Burns and Andy 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] Robert Ian Davis. *On Exploiting Spare Capacity in Hard Real-Time Systems*. PhD thesis, University of York, 1995.

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

[4] John P. Lehoczky and Sandra 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.

[5] John P. Lehoczky, Lui Sha, and Jay 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.

[6] Damien Masson and Serge Midonnet. Fault tolerance with real-time java. In *WPDRTS'06 (in proceedings of the 20st IEEE International Parallel & Distributed Processing Symposium)*, page 172, Rhodes Island, Greece, April 2006.

[7] Damien Masson and Serge Midonnet. The design and implementation of real-time event-based applications with RTSJ. In *WPDRTS'07 (in proceedings of the 21st IEEE International Parallel & Distributed Processing Symposium)*, page 148, Long Beach, CA USA, March 2007.

[8] Damien Masson and Serge Midonnet. Slack time evaluation with RTSJ. In *Proceedings of the 23rd Annual ACM Symposium on Applied Computing*, pages 322–323, Fortaleza, Ceará, Brazil, March 2008. Short paper and poster.

[9] Brinkley Sprunt, Lui Sha, and John P. Lehoczky. Aperiodic task scheduling for hard real-time systems. *Real-Time Systems: The International Journal of Time-Critical Computing Systems*, 1:27–60, 1989.

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