Processing Group Parameters in the Real-time Specification for Java

A. Burns and A.J. Wellings

Department of Computer Science, University of York
YO10 5DD, U.K.
Email: {burns,andy}@cs.york.ac.uk

Abstract. *The Real-Time Specification for Java (RTSJ) does not support any particular aperiodic server technology. Instead it provides processing group parameters, which allow a group of schedulable objects to be allocated a capacity and a replenishment period. However, no guidelines are given on how to use them. Furthermore, there are currently no appropriate schedulablity analysis techniques which can be used to model the full generality of the RTSJ model. This paper proposes restrictions on the use of processing group parameters and discusses the problems of producing the associated analysis techniques.*

1 Introduction

In any system where it is required to give guarantees, aperiodic activities present a problem. As they have no well-defined release characteristics, they can impose an unbounded demand on the processor's time. If not handled properly, they can result in periodic or sporadic schedulable objects (real-time threads and asynchronous event handlers) missing their deadlines, even though those schedulable objects have been "guaranteed".

One simple way of scheduling aperiodic activities, within a pre-emptive priority-based scheme, is to run them at a priority below the priorities assigned to periodic and sporadic schedulable objects [3]. In effect, the aperiodic schedulable objects run as background activities and, therefore, cannot pre-empt the other schedulable objects. Although a safe scheme, this does not provide adequate support to aperiodic schedulable objects that will often miss their target completion times if they only run as background activities. To improve the situation, a server can be employed. Servers protect the processing resources needed by periodic and sporadic schedulable objects but otherwise allow aperiodic schedulable objects to run as soon as possible. Several types of servers have been defined by the real time community. The major ones for fixed priority based systems are: deferrable servers[5] and sporadic servers [7].

With the deferrable server, an analysis is undertaken (using, for example, the response time approach [6]) that enables a new activity to be introduced at the highest priority[1]. This process, the server, thus has a period, $\bar{T}_s$ and a

[1] Servers at other priorities are possible but the description is more straightforward if the server is given a higher priority than all the hard processes.

capacity $\bar{C}_s$. These values are chosen so that all the hard processes in the system remain schedulable even if the server executes periodically with period $\bar{T}_s$ and execution time $\bar{C}_s$. At run-time, whenever an aperiodic process arrives, and there is capacity available, it starts executing immediately and continues until either it finishes or the capacity is exhausted. In the latter case, the aperiodic process is suspended (or transferred to a background priority). With the deferrable server model the capacity is replenished every $\bar{T}_s$ time units.

The operation of the sporadic server differs from the deferrable server in its replenishment policy. With the sporadic server, if an aperiodic activity arrives at time t and uses c capacity then the server has this c capacity replenished $\bar{T}_s$ time units after t. In general, sporadic severs can furnish higher capacity than deferrable servers but have increased implementational overheads. Both can be analysed using response time analyses [2]. Sporadic servers are supported by POSIX [4].

The Real-Time Specification for Java (RTSJ) does not support any particular aperiodic server technology. Instead it provides processing group parameters, which allow a group of schedulable objects to be allocated a capacity and a replenishment period. However, no guidelines are given on how to use them and there is no indication of the appropriate schedulablity analysis techniques to be used. Indeed, to our knowledge, there are currently no appropriate schedulablity analysis techniques which can be used to model the full generality of the RTSJ model.

In section 2 of this paper, an overview of scheduling in the RTSJ is presented along with the detailed semantics of processing group parameters. Section 3 then proposes two restrictive usages: the first supports a simple deferrable server for handling aperiodic activities, the second considers groups consisting of periodic and sporadic schedulable objects. Section 4, then develops the schedulability analysis for the latter case. Finally, section 5 presents conclusions.

2 Overview of Scheduling in the RTSJ

The RTSJ incorporates the notion of a schedulable object rather than considering just threads. A schedulable object is any object that implements the `Schedulable` interface. The current specification essentially provides two types of object that implement this interface, `RealtimeThreads` and `AsyncEventHandler`s. Objects that implement the `Schedulable` interface have the following associated attributes (represented by classes).

`ReleaseParameters` - Giving the processing cost for each release (its CPU budget) of the schedulable object and its deadline; if the object is released periodically or sporadically then subclasses allow an interval to be given. Event handlers can be specified for the situation where the deadline is missed or the processing resource consumed becomes greater than the cost specified. However, note that there is no requirement for a real-time JVM to monitor the processing time consumed by a schedulable object. If it does, then there is a requirement that a schedulable object be given no more than its cost in processing units each release. It should be noted that Version 1 of the RTSJ makes no mention of blocking time in any of the parameters associated with schedulable objects. The assumption is that a particular implementation will subclass `ReleaseParameters` to bring in this data. Typically, this will be a relative time value set by the programmer as a result of off-line analysis of the code. It is likely that Version 1.1 will formally introduce blocking time into the `ReleaseParameters` class.

`SchedulingParameters` - The `SchedulingParameters` class is empty; however subclasses allow the priority of the object to be specified along with its importance to the overall functioning of the application. Although the RTSJ specifies a minimum range of real-time priorities (28), it makes no statement on the allowed values of the importance parameter.

`MemoryParameters` - Giving the maximum amount of memory used by the object in its default memory area, the maximum amount of memory used in immortal memory, and a maximum allocation rate of heap memory. An implementation of the RTSJ is obligated to enforce these maximums and throw exceptions if they are violated.

`ProcessingGroupParameters` - This allows several schedulable objects to be treated as a group and to have an associated period, cost and deadline (see Section 2.1).

The methods in the `Schedulable` interface can be divided into three groups.

- Methods that will communicate with the scheduler and will result in the scheduler either adding or removing the schedulable object from the list of objects it manages (called its feasibility set), or changing the parameters associated with the schedulable object (but only if the resulting system is feasible).
- Methods that get or set the parameter classes associated with the schedulable object. If the parameter object set is different from the one currently associated with the schedulable object, the previous value is lost and the new one will be used in any future feasibility analysis performed by the scheduler. Note, these methods do not result in feasibility analysis being performed and the parameters are changed even if the resulting system is not feasible.
- Methods that get or set the scheduler. For systems that support more than one scheduler, these methods allow the scheduler associated with the schedulable object to be manipulated.

The only scheduler that the RTSJ fully defines is a priority scheduler, which can be summarized as having:

Scheduling policy. The priority scheduler
  - supports the notion of base and active priority;
  - orders the execution of schedulable objects on a single processor according to the active priority;
  - supports a real-time priority range of at least 28 unique priorities (the larger the value, the higher the priority);
  - requires the programmer to assign the base priorities (say, according to the relative deadline of the schedulable object);

  - allows base priorities to be changed by the programmer at run time;
  - supports priority inheritance or priority ceiling emulation inheritance for synchronized objects;
  - assigns the active priority of a schedulable object to be the higher of its base priority and any priority it has inherited.

Scheduling mechanism. The priority scheduler

  - supports pre-emptive priority-based dispatching of schedulable objects - the processor resource is always given to the highest priority runnable schedulable object;
  - does not define where in the run queue (associated with the priority level) a pre-empted object is placed; however, a particular implementation is required to document its approach and the RTSJ recommends that it be placed at the front of the queue;
  - places a blocked schedulable object that becomes runnable, or has its base priority changed, at the back of the run queue associated with its (new) active priority;
  - places a schedulable object which performs a Thread.yield() method call at the back of the run queue associated with its priority
  - does not define whether schedulable objects of the same priority are scheduled in FIFO, round-robin order, or any other order.

Schedulability (feasibility) analysis. The PriorityScheduler requires no particular analysis to be supported.

2.1 Processing Group Parameters

The RTSJ provides support for aperiodic server technologies via processing group parameters. When processing group parameters are assigned to one or more schedulable objects, a logical server is created. The server's start time, cost (capacity) and period are defined by the particular instance of the parameters. These collectively define the points in time when the server's capacity is replenished.

Any aperiodic schedulable object that belongs to a processing group is executed at its defined priority. However, it only executes if the server still has capacity (and the object has not overrun its own individual CPU budget). As it executes, each unit of CPU time consumed is subtracted from the server's capacity (as well as its own). When capacity is exhausted, the aperiodic schedulable objects are not allowed to execute until the start of the next replenishment period.

The RTSJ facilities are very general. They allow

- schedulable objects of different priorities to be assigned to the same group,
- the inclusion of sporadic and periodic schedulable objects,
- the schedulable objects to also have individual costs and deadlines,
- the "servers" to be given a deadline, and cost overrun and deadline miss handlers.

The latter may be useful if the programmer has more information about the pattern of arrivals of the aperiodic activities and wants to detect variations from that pattern.

The full definition of the `ProcessingGroupParameters` class is given below.

```
package javax.realtime;
public class ProcessingGroupParameters {
  // constructors
  public ProcessingGroupParameters( HighResolutionTime start,
         RelativeTime period, RelativeTime cost,
         RelativeTime deadline,
         AsyncEventHandler overrunHandler,
         AsyncEventHandler missHandler);

  // methods
  public RelativeTime getCost();
  public AsyncEventHandler getCostOverrunHandler();
  public RelativeTime getDeadline();
  public AsyncEventHandler getDeadlineMissHandler();
  public RelativeTime getPeriod();
  public HighResolutionTime getStart();
  public void setCost(RelativeTime cost);
  public void setCostOverrunHandler(AsyncEventHandler handler);
  public void setDeadline(RelativeTime deadline);
  public void setDeadlineMissHandler(AsyncEventHandler handler);
  public void setPeriod(RelativeTime period);
  public void setStart(HighResolutionTime start);
}
```

3 Using Processing Group Parameters

Processing group parameters allow very general servers to be set up. However, to get full predictability, it will be necessary for implementations to restrict the generality so that they support one (or more) of the well-known server technologies, such as deferrable servers. This section considers the various ways in which processing group parameters can be used.

3.1 Handling Aperiodics

The simpliest form of restriction on processing group parameters is to provide a subclass of `ProcessingGroupParameters` so that the scheduler can support, say, deferrable servers.

```
import javax.realtime.*;
public class DeferrableProcessingGroupParameters extends
             ProcessingGroupParameters  {

  public DeferrableProcessingGroupParameters(
```

```
        HighResolutionTime start, RelativeTime period,
        RelativeTime cost, PriorityParameters pri) {
    super(start, period, cost, period, null, null);
    // save priority etc, throw exceptions
  }
  ...
}
```

The scheduler would then throw exceptions if schedulable objects having the same deferrable processing group parameters do not have the same priority and do not have aperiodic release parameters. Standard schedulability analysis can be used to model the impact that a particular group will have on the rest of the system.

Note also, that cost overruns and deadline misses of the individual aperiodic activities will also be detected.

The advantages of this approach include: it models one of the standard server technologies, it is simple to implement, and it uses standard analysis techniques.

The disadvantage is that aperiodic activities are handled, effectively, in FIFO order.

A simple extension of the above allows the aperiodic activities to have different priorities. However, given that there are no bounds on the release of each of the aperiodic schedulable objects, the analysis must assume that the highest priority one is always released. Hence, the server is considered to run at the highest priority.

3.2 Handling Periodic and Sporadics

Whilst it might seem curious to want to handle periodic activities with servers, it is useful for the situation where there is a subsystem containing some periodic schedulable objects which have a large variation between their average and worst case execution time. In this situaton, assuming that each object performs to its worst case at the same time as every other object of that system is very pessimistic. Using processing group parameters it is possible to

- bound the interfence of the group to other schedulable objects in the system, thereby ensuring that the latter will not be impacted if the worst case phasing does occur
- allow each schedulable object to obtain its maximum CPU time when it needs it.

Including sporadic schedulable objects into a processing group ensures that they do not occur more frequently than estimated.

Of course, with this approach it may not be possible to give guarantees to all the schedulable objects in the group. However, it is possible to calculate their individual response times and the impact that the group has on other schedulable objects in the system. See section 4.

The following class illustarted how subsystems might be represented.

```java
import javax.realtime.*;
public class SubsystemProcessingGroupParameters extends
             ProcessingGroupParameters
{

  public SubsystemProcessingGroupParameters(
        HighResolutionTime start,
        RelativeTime period, RelativeTime cost)
  {
    super(start, period, cost, period, null, null);
  }
   ...
}
```

4 Response-Time Analysis for Subsystem Group Parameters

The analysis in this section assumes that the application under consideration consists of a set of periodic or sporadic schedulable objects [2], e.g. τ_i. Each τ_i has a minimum inter-arrival time of T_i, maximum computation time, C_i, and deadline D_i ($D_i \leq T_i$). We need to verify that the worst-case response-time, R_i is always less than its deadline ($R_i \leq D_i$). We assume that each τ_i has a unique priority P_i and is a member of exactly one server group [3]. Each server S_s has a capacity $\bar{C}_s$ and a replenishment period $\bar{T}_s$. For simplicity we assume each server's deadline is equal to $\bar{T}_s$. We also assume that τ_i does not share data with any object in another server (although it may do so with schedulable objects within its own server).

If, as will be the case in general, τ_i executes over a number of server periods it is not possible to calculate R_i exactly (it is a NP-hard problem). This is due to (at least) two properties of the computation model:

– The worst-case does not necessary occur when all higher priority objects are released at the same time (the critical instance).
– The worst-case does not necessarily occur when sporadic objects arrive at their maximum frequency.

Both of these properties make the problem of calculating an exact R_i intractable. In the following analysis therefore an approximation is produced that gives an upper-bound for R_i. This analysis allows the feasibility of an application to be tested and, perhaps more importantly, allows the designer to explore different values for the server parameters ($\bar{C}_s$, $\bar{T}_s$).

[2] It is trivial to include aperiodic schedulable objects represented by deferrable servers into this model.

[3] A schedulable object which is not a member of a group is modeled as the sole member of a group whose release parameters are identical to the schedulable object's release parameters.

Although the parameter $\bar{C}_s$ is described as the server's capacity it is not guaranteed. It may be the case that S_s is never allocated that amount of execution time in any $\bar{T}_s$. Note that any capacity not used in $\bar{T}_s$ is lost. What $\bar{C}_s$ represents is the maximum capacity that the server is allowed to assign to the objects it serves. To calculate R_i, we need to compute the guaranteed capacity each server ($\tau_i \in S_s$) will obtain in any $\bar{T}_s$. As the server executes at the priority of its clients we need to obtain $\hat{C}_s(i)$, the guaranteed capacity of S_s at priority level P_i (note $\hat{C}_s(i) < \bar{C}_s$ - it is capped by the server's capacity).

To evaluate $\hat{C}_s(i)$ we need to calculate the maximum interference from other servers and other objects within S_s. First other servers: analysis of deferred servers [2] indicate that the worst-case occurs when $\hat{C}_s(i)$ is available for interference at the very end of a server's period and again at the start of the following period. This is illustrated in Figure 1.

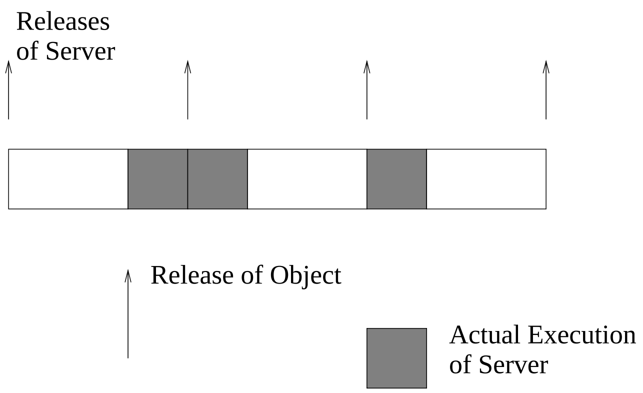

Fig. 1. Worst-Case Release of External Server

If the server (S_ρ) was always interfering with τ_i then in any interval $[0\ t)$ the interference would be

$$\left\lceil \frac{t + (\bar{T}_\rho - \bar{C}_\rho)}{\bar{T}_\rho} \right\rceil \bar{C}_\rho \tag{1}$$

But this is an upper-bound. Server S_ρ will only interfere if it has objects with higher priority than P_i.

The usual response time equation has the following term for interference:

$$\sum_{\tau_m \in hp(i,\rho)} \left\lceil \frac{t}{T_m} \right\rceil C_m \tag{2}$$

where $hp(i,\rho)$ is the set of schedulable objects with priority greater than P_i in server S_ρ .

Combining (1) and (2) we get a formulae for the interference from server S_ρ ($\tau_i \notin S_\rho$):

$$I_s(i,t) = \min\left(\left\lceil \frac{t + (\bar{T}_\rho - \bar{C}_\rho)}{\bar{T}_\rho} \right\rceil \bar{C}_\rho \ , \sum_{\tau_m \in hp(i,\rho)} \left\lceil \frac{t}{T_m} \right\rceil C_m \right) \tag{3}$$

The usual response time equation is solved for a particular computation time C. In this formulation we wish to obtain the maximum c that will be guaranteed in $\bar{T}_s$. That is

$$r = c + \sum I_s(i,r) \tag{4}$$

where the sum is over all the servers apart from S_s ($\tau_i \in S_s$).

This is solved to give $r \leq \bar{T}_s$ - but $(c+1)$ would produce a value of r greater than $\bar{T}_s$. There are various branch and bound methods of iterating towards this value of c [1]. Once c is calculated then this is the required value of $\hat{C}_s(i)$.

To return to the problem of calculating the response time R_i of object τ_i we need to allocate $\hat{C}_s(i)$ to the requirements of τ_i, namely C_i. Unfortunately in the worst-case the full capacity of the server S_s has been used up just before τ_i arrives, and further $\bar{C}_s(i)$ amounts are only available at the end of the server's replenishment period. This is illustrated in Figure 2.

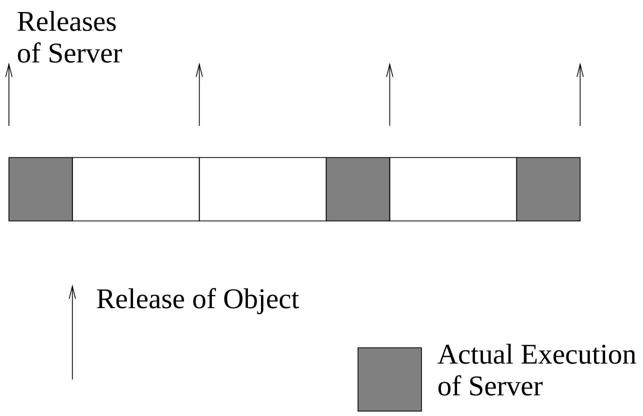

Fig. 2. Worst-Case Release of Own Server

The final issue to note is that data sharing via some appropriate priority inheritance protocol will mean that $C_i + B_i$ (where B_i is the blocking term in the standard analysis) needs to be guaranteed. So let V be defined by

$$V = \left\lceil \frac{C_i + B_i}{\hat{C}_s(i)} \right\rceil \tag{5}$$

then

$$R_i \leq (\bar{T}_s - \bar{C}_s) + V\bar{T}_s \tag{6}$$

This is a safe upper bound. Some improvements can be obtained by:

– Being more exact about the maximum capacity that could have been removed prior to being released (as depicted in Figure 2) - for low priority objects there may not exist even lower priority objects to 'steal' the capacity.
– Also, as illustrated in Figure 2, the guaranteed capacity may not be right at the end of the period - the value of r in equation (4) would give a tighter completion time.
– The amount of guarantee required in the final period is likely to be less than a full $\hat{C}_s(i)$ - again solving equation (4) for the actual c required in this last period would give a tighter completion time.

All of these improvements will reduce the bound on R_i, but to reiterate the point made at the beginning of this section, a (general) tight bound on R_i is not possible to obtain. An analysis of specific application requirements is needed to judge the effectiveness of the analysis represented here.

5 Conclusions and Future Work

The RTSJ has introduced a novel approach to capacity sharing between schedulable objects. In its full generality, the model is too complex to develop supporting analysis techniques. This paper has proposed two restrictive usages. One for handling groups of aperiodic activities and the other for handling groups of periodic and sporadic activities. The former can be supported by standard deferrable server analysis techniques. The latter needs new analysis techniques. This paper has discussed the problems of developing these techniques and goes some way to solving them. Future work will include: verifying the approach by simulation and extending the model to allow for communication between schedulable objects in different groups.

Acknowledgement

The authors gratefully acknowledge the contribution of Guillem Bernat to some of the ideas expressed in this paper.

References

1. G. Bernat. Response time analysis of asynchronous systems. *Real-Time Systems*, to appear.

2. G. Bernat and A. Burns. New results on fixed priority aperiodic servers. In *Proceedings of the 20th IEEE Real-Time Systems Symposium*, pages 68–78, 1999.
3. A. Burns and A. J. Wellings. *Real-Time Systems and Programming Languages:*. Addison Wesley, 3rd edition, 2001.
4. IEEE. Portable operating sysrem interface: Amendment d: Additional real-time extension [C language]. IEEE/1003.1d, IEEE, 1999.
5. J. P. Lehoczky, L. Sha, and J. K. Strosnider. Enhanced aperiodic responsiveness in a hard real-time environment. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 261–270, 1987.
6. J.Y.T. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation (Netherlands)*, 2(4):237–250, 1982.
7. B. Sprunt, J. Lehoczky, and L. Sha. Exploiting unused periodic time for aperiodic service using the extended priority exchange algorithm. In *Proceedings IEEE Real-Time Systems Symposium*, pages 251–258, 1988.