Asynchronous Event Handling and Real-time Threads in the Real-time Specification for Java*

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

Abstract

This paper discusses the role and implementation of asynchronous event handlers in the Real-time Specification for Java (RTSJ). For non-blocking handlers, an implementation model whereby all heap-using handlers are serviced by a single thread and all no-heap handlers are serviced by another server thread is proposed and schedulability analysis is derived. The model is shown to have bounded priority inversion. General multiple-server models are needed for non-blocking handlers but the support in the RTSJ is criticised as lacking in configurability. A solution is proposed which allow the number of servers to be specified, and the allocation of handlers to servers to be controlled.

1 Introduction

One of the main reasons for a real-time programming language to support concurrency is to facilitate the modelling of parallelism in the real world (Burns and Wellings, 2001). For example, within embedded system design, the controllers for real world objects such as conveyor belts, engines and robots are represented as threads in the program. The interaction between the real world objects and their controllers can be either time triggered or event triggered. In a time triggered systems, the controller is activated periodically. It senses the environment in order to determining the status of the real-time object it is controlling. Based on its findings, it writes to actuators which are able to affect the behaviour of the object. For example, a robot controller may determine the position of a robot via a sensor and decide that it must cut the power to a motor thereby bringing the robot to a halt. In an event triggered system, sensors in the environment are activated when the real world object enters into certain states. The events are signalled to the controller via interrupts. For example, a robot may trip a switch when it reaches a certain position. This is a signal to the controller that the power to the motor should be turn off, thereby bringing the robot to a halt.

The system designer often has a choice on whether to implement the control algorithm as a time triggered or event triggered one. Event triggered systems are often more flexible whereas time triggered system are more predictable (Kopetz, 1997; Burns, 2002). In either case, the controller is usually represented as a thread. However, there are occasions where this is not appropriate. These include (Ousterhout, 2002; van Renesse, 1998) when:

- the external objects are many and their control algorithms are simple and non blocking, and
- the external objects are inter-related and their collective control requires significant communication and synchronization between the controllers.

In the former case, using a thread per controller leads to a proliferation of threads along with the associated per thread overhead. In the latter case, complex communication and synchronization protocols are needed which can be difficult to design correctly and may lead to deadlock or unbounded blocking.

An alternative to thread-based programming is event-based programming. Each event has an associated handler. When events occur, they are queued and a server thread takes an event from the queue and executes its associated handler to completion. When the handler has finished, the server takes another event from the queue, executes the handler and so on. The execution of the handlers may generate further events. With this model, there is only one thread – the server thread. There is no need for explicit communication between the handlers as they can simply read and write from shared objects with-

*Java is a trademark of Sun Microsystems.

out contention. The disadvantage of controlling all external objects by event handlers include:

- it is difficult to have tight deadlines associated with event handlers as a long-lived or blocking handler must terminate before the server can execute any newly arrived high-priority handlers;
- it is difficult to execute the handlers on a multiprocessor system as the handlers assume no contention for shared resources.

One of the main examples often quoted as requiring an event-handling system is the implementation of a graphical user interface. For example, standard Java supports threads but its Swing and AWT toolkits are event based.

In an attempt to provide the flexibility of threads and the efficiency of event handling, the Real-Time Specification for Java (RTSJ) (Bollella et al., 2000) has introduced the notion of real-time asynchronous events and associated handlers. However, the specification is silent on how these events can be implemented and how their timing requirements can be guaranteed. As the RTSJ is likely to become a standard implementation language for real-time applications, a detailed assessment of its facilities is timely. The goal of this paper is to explore how RTSJ real-time events can be implemented and integrated with the scheduling of real-time threads. In section 2, and overview of the RTSJ event handling model is presented. The only scheduling policy required by RTSJ is priority-based scheduling; section 3, therefore, considers a priority-based implementation of event handlers. Section 4 then develops the response time analysis for this implementation model. Section 5 extends the model by considering multiple servers and bound event handlers. Finally, conclusions are presented in Section 6.

2 The Event Handling Model

Objects which are to be scheduled in RTSJ must implement the `Schedulable` interface and specify:

- their memory requirements via the class `MemoryParameters` – not considered in this paper;
- their timing requirements via the class `ReleaseParameters`;
- their scheduling requirements via the class `SchedulingParameters`;

The `ReleaseParameters` class is a base abstract class and gives the general parameters that all schedulable objects need. A schedulable object can have a deadline and a cost associated with each time it is released for execution, along with handlers for their overrun. The cost is the amount of execution time that a scheduler should give to the object. If the object is still executing when either its deadline or its cost expires, an associated event handler is scheduled. It should be noted that the RTSJ does not require an implementation to support execution-time monitoring. However, it does require it to detect missed deadlines. Of course, a program can indicate that it is not concerned with a missed deadline by passing a null handler. Subclasses of the `ReleaseParameters` class support periodic, aperiodic and sporadic release parameters.

The `SchedulingParameters` is a null abstract class. It has subclasses for the definition of priority parameters and important parameters. All RTSJ implementations are required to support at least 28 unique real-time priority levels. However, there is no defined values for the importance parameters. The only scheduler that the RTSJ requires is a preemptive priority-based one.

Essentially there are two types of schedulable objects: real-time threads and asynchronous event handlers. Each `AsyncEvent` can have one or more handlers. When the event occurs (indicated by a call to the `fire` method), all the handlers associated with the event are scheduled for execution according to their `SchedulingParameters`. Note that the firing of an event can also be associated with the occurrence of an implementation-dependent external action by using the `bindTo` method. Subclasses of the `AsyncEvent` class provide time-triggered events. Figure 1 illustrates the event class hierarchy.

Each handler is scheduled once for each outstanding event firing. However, the handler can modify the number of outstanding events by using the methods in the `AsyncEventHandler` class.

Although an event handler is a schedulable entity (i.e. at some point it must be executed by a thread), the intention of the RTSJ is that it will not suffer the same overhead as an application thread. Consequently, it cannot be assumed that there is a separate implementation thread for each handler, as more than one handler may be associated with a particular implementation thread. If a dedicated thread is required, the `BoundAsyncEventHandler` should be used. However, other event handlers may also be associated with that thread. Figure 2 illustrates the event handler class and its relationship to other classes.

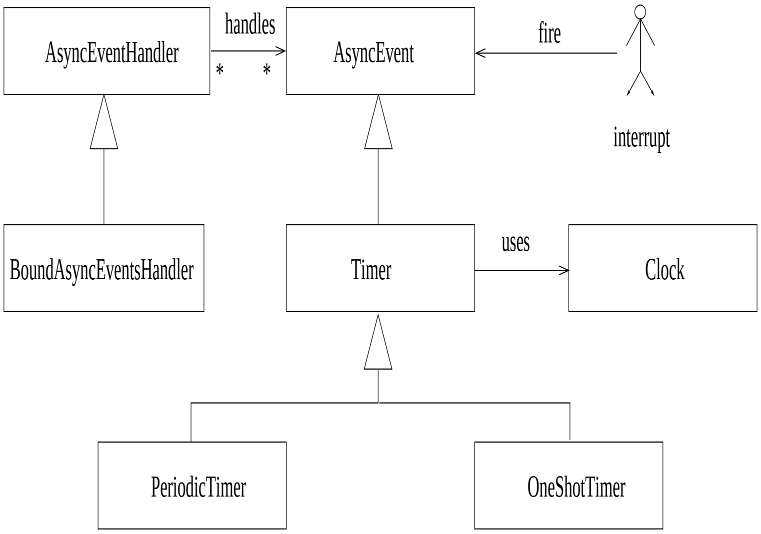

Figure 1: Event class in RTSJ

3 Implemention Issues

The key challenge in implementing event handlers is to limit the number of threads without jeopardising the schedulability of the overall system. Furthermore, the RTSJ requires pre-emptive priority-based scheduling of all schedulable objects. If interpreted strictly, this requires a high priority event handler to pre-empt a low priority event handler. This severely limits the freedom of the implementation and, arguably, removes one of the motivations for an event handling paradigm (i.e. that communicating handlers are non-pre-emptible and, therefore, do not require synchronization).

This section considers only sporadic and periodic events handlers; aperiodic event handlers can be implemented using standard aperiodic server technology such as sporadic servers or deferrable servers (Lehoczky et al., 1987). From a scheduling perspective, each of these handlers has an inter-arrival time, a deadline and a priority. Here it is assumed that the priorities have been set by the programmer, perhaps using a deadline monotonic priority assignment algorithm (Leung and Whitehead, 1982).

At one extreme of the possible implementation strategies is the approach which allocates a distinct thread (real-time or no-heap real-time, depending on the handler) for each handler and schedules the threads in competition with application-defined threads. Whilst this is simple to implement and is adequate for systems with a small number of events, it is expensive for a large number of events.

At the other extreme, a single server real-time thread could be used to execute all event handlers. This approach is illustrated in Figure 3.

There are two main issues associated with this approach:

1. the order of the event handler queue
2. the priority of the server

To enable effective schedulability analysis, the event handler queue should be priority ordered. Note that for event firing, zero, one or more handlers may be added to the queue. The handlers might have different priorities. It is also necessary to check that the inter-arrival time of the release of the handlers has not been violated; however, this is not the concern of this paper.

The priority of the server needs to reflect the priority of the handler it is executing. Hence, the server's priority must be dynamically changed. Furthermore, as an executing low priority handler is not pre-emptible by a high priority handler (in this model), it is necessary to implement a priority inheritance algorithm in order to get usable bounds for the priority inversion. The priority of the server can, therefore, be defined to be the maximum of the priority of the handler it is currently executing and the priority of the handler at the front of the event handler queue*. Note that although the priority of the server is changing, it is still essentially a fixed priority system albeit with priority inheritance.

*A similar approach can be used if EDF scheduling is employed instead of priority-based scheduling.

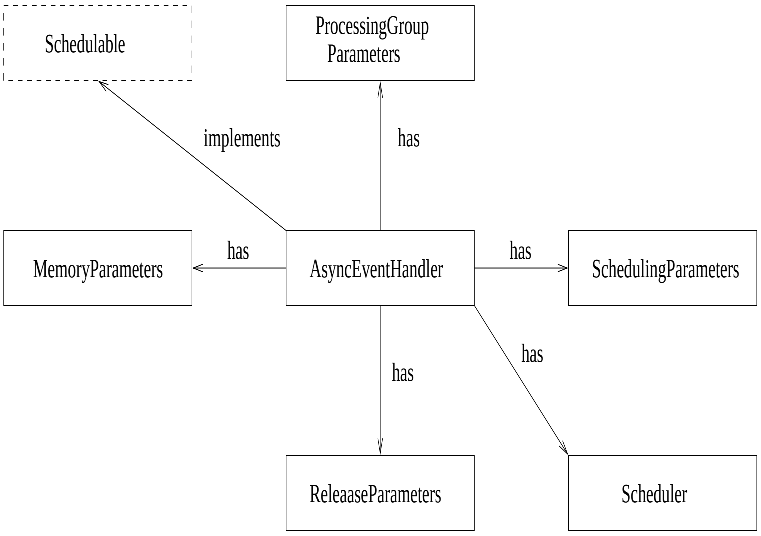

Figure 2: Event handler class in RTSJ

Unfortunately, there are some significant disadvantages with this approach.

1. Potentially Unbounded Priority Inversion

   The server causes priority inversion. A high priority event handler will not be executed immediately. Instead it is blocked until the server finishes executing the current handler. Priority inheritance allows this blocking to be bounded (see Section 4) but only if the handlers themselves do not block (e.g., issue a sleep or wait method call). Multiple servers are needed to avoid this problem (see Section 5).

2. No-Heap Event Handlers

   The RTSJ allows real-time threads and asynchronous event handlers to indicate that they will not access heap memory. This enables them to safely pre-empt the garbage collector. The single server model presented above will fail when there is a mixture of heap and no-heap event handlers. For example, the server executing an event handler which uses the heap can inherit a no-heap event handler's priority. However, the thread is a real-time thread and, therefore, can still be pre-empted by the garbage collector. Consequently, the no-heap handler will be delayed when garbage collection occurs. The solution to this problem (when handler do not block) is to have two server threads (a real-time thread and a no-heap real-time thread) and two priority ordered queues (one for real-time handlers and one for no-heap handlers).

4 Response Time Analysis

For fixed priority systems, response time analysis can be used to determine whether the set of threads meet their deadlines (Audsley et al., 1993). Table 1 gives a standard set of notations used for analysing these systems. The response time equation for an arbitrary thread i is given below:

$$R_i = C_i + B_i + I_i$$

that is,

$$R_i \;=\; C_i \;+\; B_i \;+\; \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \qquad (1)$$

which can be solved by constructing a recurrence relationship:

$$R_i^{n+1} \;=\; C_i \;+\; B_i \;+\; \sum_{j \in hp(i)} \left\lceil \frac{R_i^n}{T_j} \right\rceil C_j \qquad (2)$$

the recurrence iteration starts with R_i^0 equal to C_i.

This equation needs to be modified to capture the model of event handling presented in Section 3 (for simplicity the analysis is shown for just one real-time server, i.e. there is no mixture of heap and no-heap handlers, and there are no delays from garbage collection). Furthermore, the analysis assumes that event handlers do not suspend themselves.

The approach is to model each event handler as an individual thread with its own computation time,

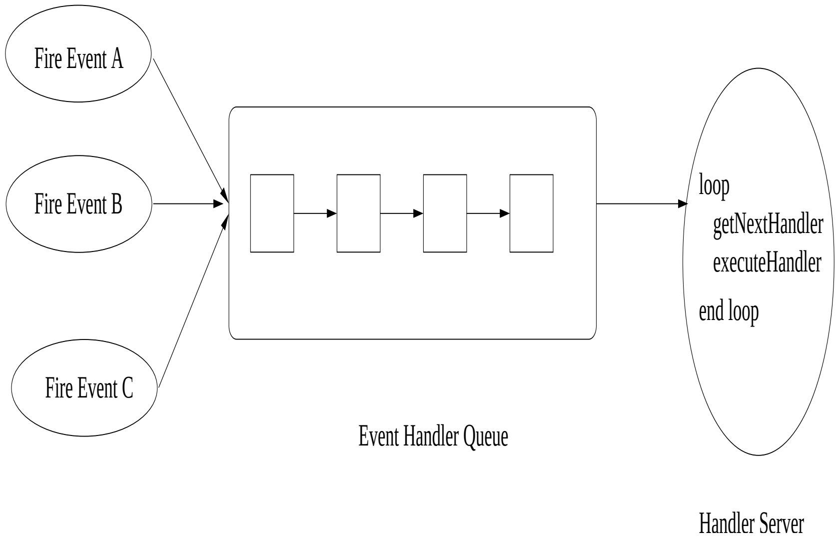

Figure 3: Handling events via a single server

Notation	Description
B	Worst-case blocking time for the thread (if applicable)
C	Worst-case computation time (WCET) of the thread
D	Deadline of the thread
I	The interference time the thread suffers from higher priority threads
N	Number of threads in the system
P	Priority assigned to the thread (if applicable)
R	Worst-case response time of the thread
T	Minimum time between thread releases (e.g. thread period)
$hp(i)$	The set of threads with priority higher than thread i

Table 1: Standard notation

however, the non-pre-emptible nature of their execution has to be taken into account. Consider the response time of an arbitrary thread/handler. Its interference occurs from high priority threads plus higher priority event handling. There is also an added component from the non-pre-emptive time of the server. This is the maximum of all low priority event handlers. Even for threads, this non pre-emption time must be included even though they do execute pre-emptively. To illustrate this consider a thread that pre-empts a lower priority handler. Then let a high priority hander be released. It will pre-empt the thread but it cannot execute until the lower priority handler has completed. Priority inheritance raises the priority of the original handler so that it does execute before the thread. Hence the execution time of lower priority handlers must be included for threads as well as other handlers.

$$\begin{aligned} I_i &= \textstyle\sum_{j \in hpt(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j + \\ & \textstyle\sum_{l \in hph(i)} \left\lceil \frac{R_i}{T_l} \right\rceil C_l + \max_{k \in lph} C_k \end{aligned} \tag{3}$$

where $hpt(i)$ is the set of threads whose priority is higher than thread/handler i and $hph(i)$ is the set of handlers whose priority is higher than thread/handler i. $lph(i)$ is the set of handlers whose priority is lower than or equal to thread/handler i. Hence, the worst-case non-pre-emption time is the maximum of all the low priority handlers.

The above is an adequate conservative model and provides a reasonable tight bound on the response time for non-blocking handlers if the execution time of handlers is reasonable short. A tighter result is obtained by noting that a handler cannot be pre-empted by a higher priority handler once it has started executing (although it can be pre-empted by a higher priority thread). To account for this we must first calculate the worst case response time (R^{Δ}) for executing the first 'tick' (Δ)of the handler:

$$R_i^{\Delta} = \Delta + B_i + \sum_{j \in hpt(i)} \left\lceil \frac{R_i^{\Delta}}{T_j} \right\rceil C_j + \sum_{l \in hph(i)} \left\lceil \frac{R_i^{\Delta}}{T_l} \right\rceil C_l + \max_{k \in lph} C_k \quad (4)$$

This is solved using the recurrence method. The full response time is now obtained by a minor change to this equation (note no more interference is obtained from the handlers).

$$R_i = C_i + B_i + \sum_{j \in hpt(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j + \sum_{l \in hph(i)} \left\lceil \frac{R_i^{\Delta}}{T_l} \right\rceil C_l + \max_{k \in lph} C_k \quad (5)$$

to solve this, the recurrence equation starts with the value R_i^{Δ}.

4.1 Modelling Overheads

Equations (3) and (5) have two main limitations

1. they does not take into account context switch times, and therefore, part of the motivation for event handlers is lost
2. they does not model the time taken to put/take the event handlers into/from the queue.

With normal pre-emptive scheduling of threads, it is usual to account for the overheads of context switching by adding to the execution time of each thread the cost of switching to, and the cost of switching from, the thread. Although the scheduling analysis assumes that the worst case occurs when all higher priority threads are released together, once context switches are incorporated, the worst case actually occurs when each higher priority thread pre-empts the thread under consideration.

With the non-pre-emptive handler model described above it would appear that less context switches will occur (as all handlers are being executed by the same thread). Indeed the average number of context switches will be much lower than one would observe if each handler had its own thread. However in the worst case, the single thread model behaves no better than the multi-thread approach[†]. To understand this, consider a handler of priority p with an application thread τ of priority $p+1$. Clearly τ runs before the handler; but during its execution other higher priority handlers will arrive and pre-empt τ. Each of these will result in two context switches – and hence the handler under consideration will suffer these overheads. If there are no application threads with priorities interleaved with the handler priorities then it may be that a reduced overhead load can be used in the analysis – but this will depend on the particular characteristics of the application.

[†]There is some reduction in overhead due to non-pre-emption, but this is already taken care of in equation(5).

Events are usually associated with interrupts and hence placing the event handlers in the queue can be considered to be performed at the highest priority. Furthermore, as the queue must be protected, the server must access it at the highest priority and ceiling priority inheritance used. The additional interference these interrupt handlers have on each application threads is given by:

$$\sum_{k \in \Gamma} \left\lceil \frac{R_i}{T_k} \right\rceil IH$$

where Γ is the set of handlers, IH is the cost of handling the interrupt (and returning to the running thread, having entered the associated handlers into the queue and, if necessary, raising the priority of the server). It can also include the cost of taking the associated event handlers out of the queue.

This representation assumes that all interrupt handlers give rise to the same cost; if this is not the case then IH must be defined for each k.

5 Multiple Servers and Bound Event Handlers

The RTSJ allows an implementation of asynchronous event handlers to use more than one server thread and for there to be a dynamic association of handlers to threads. A possible model is depicted in Figure 4:

This general model has two main drawbacks:

- related handlers must now assume that there may be some contention for shared software resources
- the worst-case response times of the handlers is not dramatically improved as no knowledge of handler to server allocation can be assumed. The non-pre-emption time, say for a three server system, is the minimum of the three maximum values of the lower priority event handlers. To avoid this priority inversion, a new server thread could be created everytime the priority of the handler at the head of the queue is greater that the priorities of the current servers. However, this more dynamic approach is more difficult to analyse.

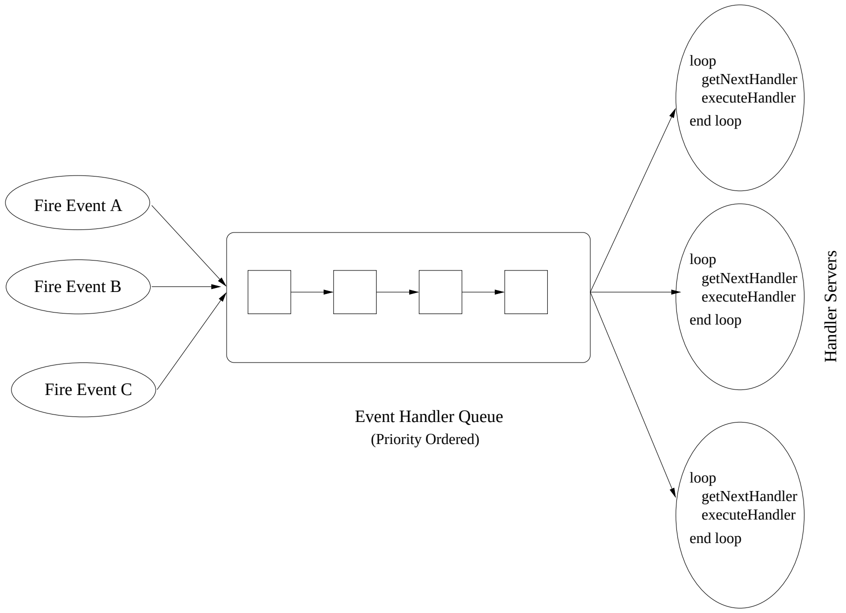

Figure 4: Handling events via multiple servers

The main advantages of this approach over the single server one is that:

- it has a better mapping for a multiprocessor system
- if a handler blocks waiting for I/O (or calls the sleep or wait methods), other handlers may still be served. However, if all handlers block, significant priority inversion still occurs.

Bound asynchronous event handlers extend this model by requiring that the same server always executes the handler. The argument for this is to reduce the latency on allocating a server. However, in the general case this would appear to have negligible benefit. Of course, in the extreme case where only one handler is bound to a particular handler, the model is equivalent to having an explicit real-time thread.

5.1 Multiple Queues and Multiple Handlers

A more flexible model can be obtained by having multiple queues of executable handlers, as depicted in Figure 5. Here, the scheduler is responsible for deciding which handler should be placed in which queue according to any feasibility analysis it can perform. One possibility is to have a queue per priority level (Dibble, 2002) where each queue has an associated server. If the handlers do not block, the analysis model is quite simple but pessimistic. To analyse the response time of a particular handler, it must be assumed that all other handlers at the same priority level are ahead of it in the queue. Again, to circumvent unbounded priority inversion when the handlers block, it is necessary to dynamically create new servers (or take them from a pool) as, and when, necessary.

The main disadvantage of this approach is that the application's programmer is not aware of the mapping; and consequently it must still assume that there may be contention for shared resources between handlers. Furthermore, the model would seem to violate the requirements for bound event handlers.

5.2 Obtaining More Control on Event Handling Implementations

In order to obtain the full benefit of asynchronous event handling, the programmer needs to be able to exercise more control over the allocation of handlers to server threads. One way of achieving this is to allow the programmer to specify the maximum number of server threads for bound event handlers. Each server thread can then be given an id. For example, consider an addition to the `RealtimeSystem` class

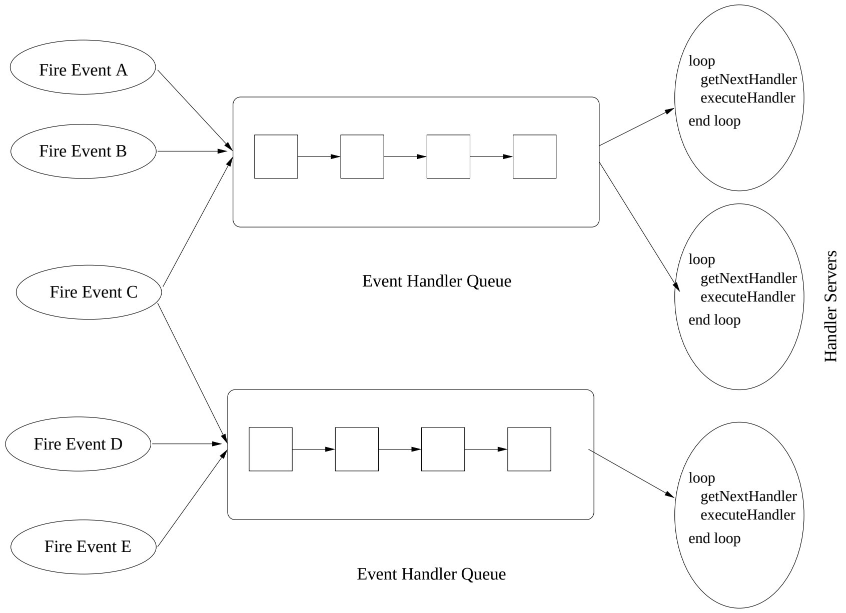

Figure 5: Handling events via multiple queues and multiple servers

```
package javax.realtime;
public class RealtimeSystem
{
  ...
  public int getNoBoundEHServers();
  public void setNoBoundEHServers(
                       int num);
  ...
}
```

Servers are numbered from `0 .. getNoBoundEHServers() - 1`. The constructors for a `BoundEventHandler` can then be modified to take an optional parameter indicating which server to use. This allows the programmer to ensure that, for example:

- handlers in separate servers do not need to synchronized their actions;
- handlers with tight deadlines can be kept separate from handlers with long execution time;
- handlers which block can be allocated to individual threads;
- heap and no-heap handlers can be bound to separate threads.

Analysis on how to find the minimum number of handlers and how to map deadline requirements to these handlers can be obtained by apply the approach used in standard pre-emptive scheduling in defining non-pre-emption groups - see (Wang and Saksena, 1999; Davis et al., 2000; Saksena and Wang, 2000).

6 Conclusions

This paper has considered the rationale for, and the implementation of, sporadic and periodic asynchronous event handlers in the Real-Time Specification for Java. Although there are no restrictions placed on the structure or content of event handlers, to obtain their full benefit, the code of handlers should be kept as short as possible and should not block. This is because event handlers are implemented by server threads and one high-priority handler allocated to a server cannot pre-empt an executing low-priority handler allocated to the same server. Consequently, to avoid unbounded priority inversion, servers must be created dynamically (or allocated from a pool). Ideally, handlers which require significant computation time, or which block, should release sporadic threads which can be scheduled in a true pre-emptive manner.

An implementation model whereby all heap-using handlers are serviced by a single thread and all no-heap handlers are serviced by another server thread has been proposed and schedulability analysis has been derived. A general multiple-server model has been criticised as lacking in configurability and a solution has been proposed which allows

1. the number of servers to be specified, and
2. the allocation of bound handlers to servers to be controlled.

Such an approach has the following advantages:

- handlers in separate servers can be organised so that they do not need to synchronized their actions – thus allowing one of main benefits from event-based systems;
- handlers with tight deadlines can be kept separate from handlers with long execution time;
- handlers which block can be allocated to individual threads;
- heap and no-heap handlers can be bound to separate threads.

These advantages have to be traded against the priority inversion that is inevitably introduced when more than one handler of different priority is allocated to a single server thread. Such an approach might be formalised in the RTSJ as a new scheduler which allows the non-pre-emptive scheduling of event handlers.

7 Acknowledgements

The authors gratefully acknowledge the comments of Greg Bollella, Peter Dibble and Doug Locke on an earlier version of this paper.

References

Audsley, N. et al. (1993). Applying new scheduling theory to static priority pre-emptive scheduling, *Software Engineering Journal* **8**(5): 284–292.

Bollella, G., Brosgol, B., Dibble, P., Furr, S., Gosling, J., Hardin, D. and Turnbull, M. (2000). *The Real-Time Specification for Java*, Addison-Wesley.

Burns, A. (2002). Real-time systems, *Encyclopedia of Physical Science and Technology, Volume 14*, Academic Press, pp. 45–54.

Burns, A. and Wellings, A. J. (2001). *Real-Time Systems and Programming Languages:*, 3rd edn, Addison Wesley.

Davis, R., Merriam, N. and Tracey, N. (2000). How embedded applications using an rtos can stay within on-chip memory limits, *Proceedings for the Work in Progress and Industrial Experience Sessions, 12th EuroMicro Conference on Real-Time Systems, Royal Institute of Technology, Technical Report Number TRITA-MMK 2000*, pp. 43–50.

Dibble, P. (2002). *Real-time Java platform programming*, Sun Microsystems Press.

Kopetz, H. (1997). *Real-Time Systems: Design Principles for Distributed Embedded Applications*, Kluwer International Series in Engineering and Computer Science.

Lehoczky, J. P., Sha, L. and Strosnider, J. K. (1987). Enhanced aperiodic responsiveness in a hard real-time environment, *Proceedings of the IEEE Real-Time Systems Symposium*, pp. 261–270.

Leung, J. and Whitehead, J. (1982). On the complexity of fixed-priority scheduling of periodic, real-time tasks, *Performance Evaluation (Netherlands)* **2**(4): 237–250.

Ousterhout, J. (2002). Why threads are a bad idea (for most purposes), *http://www.home.pacbell.net/ouster/threads.ppt*, Sun Microsystems Laboratories.

Saksena, M. and Wang, Y. (2000). Scalable real-time design using premption thresholds, *Proceedings of the 21st IEEE Real-Time Systems Symposium*, IEEE, pp. 25–34.

van Renesse, R. (1998). Goal-oriented programming, or composition using events,or threads considered harmful, *Proceedings of the Eighth ACM SIGOPS European Workshop, also available at http://www.cs.cornell.edu/Info/People/rvr/-papers/event/event.ps.*

Wang, Y. and Saksena, M. (1999). Scheduling fixed priority tasks with preemption threshold, *Proceedings, IEEE International Conference on Real-Time Computing Systems and Applications*, pp. 328–335.