Refactoring Asynchronous Event Handling in the Real-Time Specification for Java

MinSeong Kim
Department of Computer Science
The University of York, UK
djmin@cs.york.ac.uk

Andy Wellings
Department of Computer Science
The University of York, UK
andy@cs.york.ac.uk

Abstract

The primary goal for asynchronous event handling (AEH) in the Real-Time Specification for Java (RTSJ) is to have a lightweight concurrency mechanism. However the RTSJ neither provides a well-defined guideline on how to implement AEH nor requires the documentation of the AEH model used in the implementation. Also the AEH API in the RTSJ are criticised as lacking in configurability as they do not provide any means for programmers to have fine control over the AEH facilities, such as the mapping between real-time threads and handlers. For these reasons, it needs the refactoring of its application programming interface (API) to give programmers more configurability. This paper, therefore, proposes a set of AEH related classes and interfaces to enable flexible configurability over AEH components. We have implemented the refactored configurable AEH API using the new specifications on an existing RTSJ implementation and this paper shows that it allows more configurability for programmers than the current AEH API in the RTSJ does. Consequently programmers are able to specifically tailor the AEH subsystem to fit their applications' particular needs.

1. Introduction

The Real-Time Specification for Java (RTSJ) augments the Java Platform with facilities for asynchronous event handling. The goal of the approach is to provide a light-weight concurrency mechanism that does not have the overheads of real-time threads but nevertheless allows event handlers to be scheduled entities. The Real-time Virtual Machine (RT-JVM) is responsible for executing application-defined handlers according to their defined scheduling parameters, typically using a pool of real-time server threads[1]. The binding between handler and server is usually (but need not be) dynamic. The expectation is that systems of hundreds of events (if not thousands) can be handled by efficient management of the thread pool. As well as supporting application-defined events, asynchronous event handlers can also be used to handle:

- external interrupts (called *happenings* by the RTSJ)
- a class of asynchronous error conditions detected by the RT-JVM such as a deadline miss or a WCET overrun of a real-time thread
- operating system signals such as those defined by the POSIX standard, for example **SIGALRM**.

Whilst the goals of the RTSJ asynchronous event handling are laudable, their realisation in the current version of the specification suffer from the following limitations.

- **A single model for all types of events handlers** – all asynchronous event must be handled in the same implementation-dependent way; it is not possible for an application to indicate a different implementation strategy for, say, interrupt handlers or non-blocking handlers.
- **Lack of implementation configurability** – the application is unable to set the size of the thread pool or to request different handlers be serviced by different pools [8].
- **A fixed view of the notion of sporadic handlers** – the RTSJ has a fixed view of the real-time properties of sporadic asynchronous event handlers requiring that each handler has *a minimum inter-arrival* time; in practice other models are also valuable, for example a maximum arrival frequency.

The above problems have led to various attempts to add more flexibility into the implementation models [8], [9]. In this paper, we contend that a simple refactoring of the RTSJ support would allow all of the above limitations to be removed. In section 2, we give a brief overview of the current asynchronous event handling API in the RTSJ and some details of how it has been implemented. In section 3 we present our refactored AEH API that provides a framework for the implementation of various event handling models. The API is backward compatible with the current RTSJ in that application will execute unchanged in the new system. In section 4, we present a case study of using the new framework to support an application-defined implementation

1. In this paper, the real-time threads which are specifically created or maintained by the system for the execution of released handlers are also called *servers* or *server threads*.

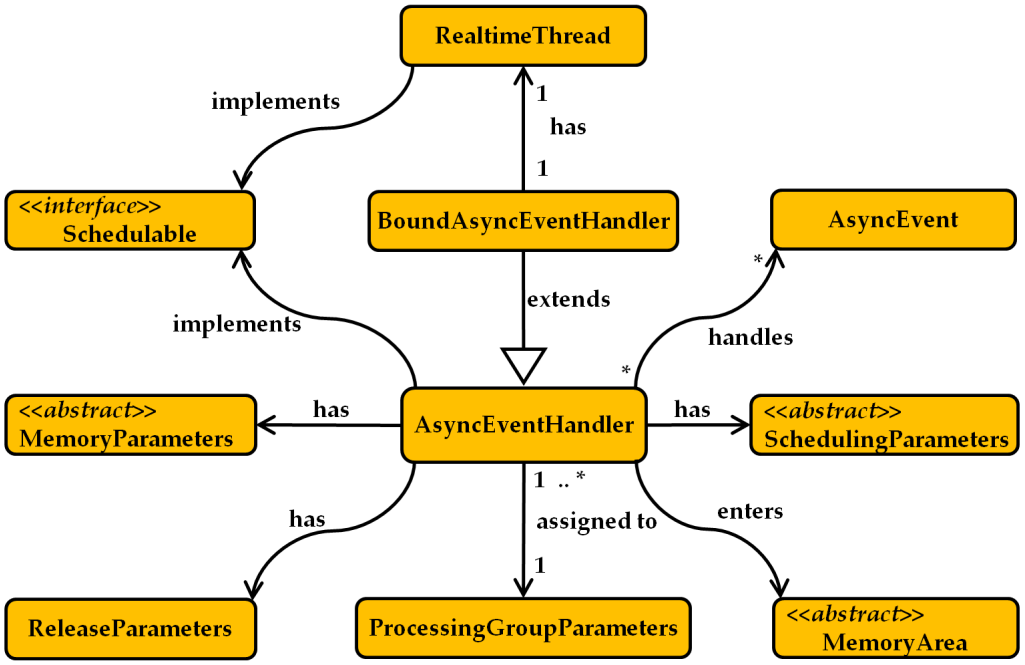

Figure 1. AEH facilities in the RTSJ

of event handlers and compare its performance to that of an implementation-defined model. Finally, section 5 presents our conclusions.

2. AEH in the RTSJ and Its Implementations

In the RTSJ, asynchronous events are viewed as dataless occurrences that can be either fired (periodically or only once) by the application or associated with the triggering of interrupts in the environment. The relationship between events and handlers is many to many (i.e. a single **AsyncEvent** can have one or more **AsyncEventHandlers**, and a single **AsyncEventHandler** can be bound to one or more **AsyncEvents**). Also handlers can be associated with POSIX signals for the situation where the RTSJ is implemented on top of a POSIX-compliant operating system.

Figure 1 illustrates the current AEH related classes and interfaces in the RTSJ. There are two types of **Schedulable** objects that implement the **Schedulable** interface: the **AsyncEventHandler** and the **RealtimeThread** class. Each instance of the two classes can therefore have a set of parameters that control its actual execution. The set of parameters specifies scheduling, timing, and memory requirements for its associated instance via **Scheduling**, **Release**, and **MemoryParameters**, respectively. Each **AsyncEvent** can have one or more **AsyncEventHandlers**. When an event is fired, all the associated handlers are scheduled for execution according to their temporal parameters. Also one or more aperiodic handlers can be assigned to a **ProcessingGroupParameters** to bound their impact on the overall schedulability of the system [16].

From the application programmers' perspective, the two **Schedulable** objects behave in the same manner. In practice, however, **RealtimeThreads** provide the vehicles for execution of **AsyncEventHandlers** [15]. In other words, **AsyncEventHandlers** will be executed by implementation-defined **RealtimeThreads** at some point. Therefore it is necessary to bind **AsyncEventHandlers** to **RealtimeThreads** and these bindings are typically performed at run-time. The time at which handlers are bound to threads and the number of threads used form the main distinguishing characteristics between AEH implementation models. If this binding latency, which inevitably incurs when the two objects are being attached to each other, is not desired **BoundAsyncEventHandlers** can be used to eliminate it. Once a **BoundAsyncEventHandler** is bounded to a **RealtimeThread**, the thread becomes the only vehicle for the execution of the particular handler, and that thread can only service that handler, no others. However the use of **BoundAsyncEventHandlers** should be minimised as dedicating a thread for each handler is very expensive for large numbers of events.

The point here is that enabling **AsyncEventHandlers** to use far fewer system resources than actual **RealtimeThreads** do, and not to suffer the same overhead as **RealtimeThreads** is the primary goal of the RTSJ. Regardless of how well the implementation is designed, as the number of **RealtimeThreads** in a system grows operating system overhead

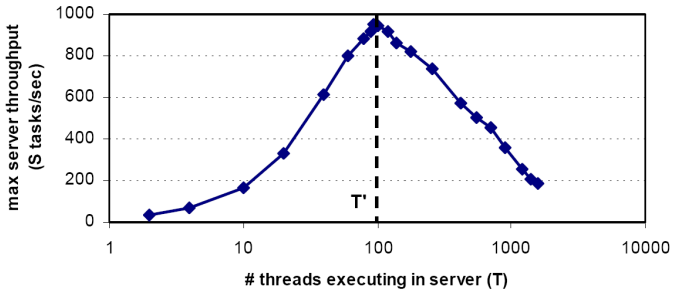

Figure 2. As the number of concurrent threads T increases, throughput increases until T = T', after which the throughput of the system degrades substantially [17]

increases, leading to a decrease in the overall performance of the system. There is typically a maximum number of threads T' that a given system can support, beyond which performance degradation occurs. This phenomenon is clearly demonstrated in Figure 2. Therefore, the key challenge in implementing AEH is to limit the number of **RealtimeThreads** without jeopardising the schedulability of the overall system. Furthermore, the resulting implementation must provide an efficient and predictable mapping between these two entities, the **AsyncEventHandler** and the server **RealtimeThread**.

2.1. AEH models in RTSJ Implementations

In [7] various AEH models used in some popular RTSJ implementations are examined and their respective pros and cons are discussed. Here we briefly review that paper's results in order to illustrate the various implementation approaches that can be adopted. The models includes the Reference Implementation (RI), OVM, Jamaica, jRate, and Java RTS. The following paragraphs summarise the run-time behavior of the AEH models:

- **RI [14]:** It creates a server each time when a handler is released and there is no active server for that handler [3]. This AEH implementation is the most intuitive way of constructing the AEH subsystem. However the scheme will likely produce a proliferation of servers along with the associated per thread overhead such as context-switches and run-time thread creation and destruction latency, when the released handlers are many.
- **OVM [6]:** Each **AsyncEventHandler** has a dedicated server permanently bound to it for the life-time of the handler. Consequently the AEH implementation of the OVM has no differences with the expected implementation of **BoundAsyncEventHandlers** of the RTSJ. It therefore does not realise the main motivation of having **AsyncEventHandlers** in the RTSJ.
- **Jamaica [5]:** The implementation tries to use fewer servers on average by allocating a dedicated server per priority level. This works well unless handlers block (self-suspend); in which unbounded priority inversion will be produced. To avoid this, it is necessary to dynamically create a new server (or take one from a pool of servers). However the implementation does not take this corrective action when required.
- **jRate [2]:** The AEH implementation in jRate uses a well-known architectural design pattern, Leader/Followers [13], that provides a concurrency model where multiple servers can demultiplex handlers and execute them. However the jRate AEH model essentially requires the same number of servers as the number of simultaneously released handlers in the system at all times. This is because the AEH implementation in jRate does not provide a facility that enables servers to execute multiple **AsyncEventHandlers**. There are also two other main drawbacks of the jRate's AEH implementation. One is that the run-time or the event-firing thread waits when all the servers in the pool are busy until one of them becomes available. This inevitably incurs unbound priority inversion if the handler, that is released when all the servers are busy, has a higher priority. The other shortcoming is that **AsyncEventHandlers** with different priorities are executed at the server's priority level in a FIFO manner as if they were assigned the server's priority [8].
- **Java RTS [11]:** This uses the late binding model for its AEH, in which notified servers defer the binding with released handlers until they are actually eligible for execution and each server is allowed to execute more than one handler. However, there are occasions when the Java RTS AEH model constantly causes unnecessary server-switching, the *Multiple Server Switching Phenomenon (MSSP)*, identified in [8]. The phenomenon can be classified as a concurrency-related issues (such as priority inversion) and takes place when the current running server changes its priority and is due to the queue replacement policy in that most real-time OSs and middle-wares put a thread that has changed its priority at the tail of the relevant queue for its new priority. Changing priorities of server threads is comprehensively used in the late binding model in order for the current running server to reflect the priority of the handler that it currently executes. Readers are referred to [8] for more detailed explanation about the phenomenon.

Figure 3 shows a sequence diagram for a simple application that uses an **AsyncEventHandler** and an **AsyncEvent**. There are two separate sections, **A** and **B**, each of which depicts the application and the implementation-specific part of the system, respectively. The programmer creates an

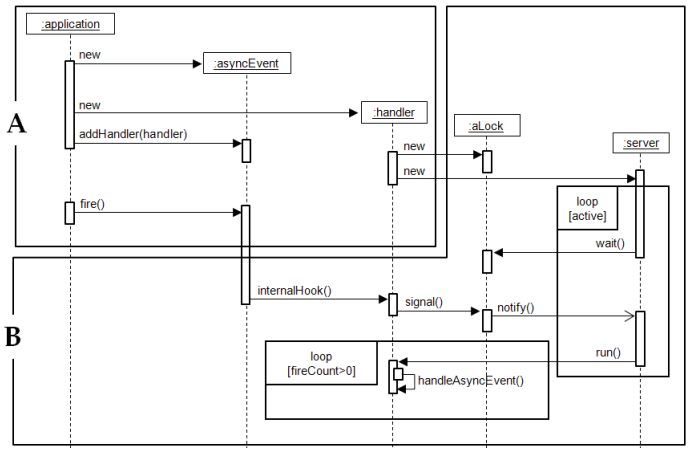

Figure 3. A Simple AEH Sequence Diagram

event and a handler. They are bound together by using the **addHandler()** method. When the event is fired (indicated by a call to the **fire()** method), the internal hook method, **internalHook()**, which causes the associated handler to be released for execution is invoked. The method in turn notifies a server or creates one depending on the AEH model currently in use. The invoked server will then execute the handler by calling the **handleAsyncEvent()** method, which holds the logic to be executed, repeatedly while the **fireCount** is greater than 0. **fireCount** is designed to remedy overload problems when handlers are being heavily used (i.e. event bursts - handlers that release before the previous release has not completed). By correctly implementing and using the concept of **fireCount**, associated overhead such as starting and stopping the server thread in which a handler run can be avoided.

The AEH models discussed in this section do not provide configuration facilities for programmers to have fine control over the AEH components and to regulate the run-time behavior. Furthermore most do not provide adequate documentation for their respective run-time behavior. The following section will present a AEH API for the RTSJ, that enables configurability for application programmers.

3. Refactoring the AEH API in the RTSJ

In Section 2.1 the overall design of the RTS AEH facilities were presented. The design is based on two premises:

1) that all handlers should be schedulable objects, and
2) that the application need not be concerned with how AEH are executed to meet their timing requirements.

In this section we remove these premises and refactor the API to extract out the notion of a *handleable* object and define the interface between the firing of an event and the release of a handler. By doing so, we maintain compatibility with the current API (from the programmer's perspective) and allow greater configurability and flexibility.

The refactored AEH API hierarchy is shown in Figure 4. The **Handleable** interface captures the properties of any object that wishes to handle an asynchronous event. It also extends the **Runnable** interface like **Schedulable** does as it should be implemented by any class whose instances are intended to be executed by a thread. It declares two additional methods, **fired()**, and **handleAsyncEvent()**. The former method is declared here to explicitly define the relationship between the **AsyncEvent** and the **Handleable** class. The latter method was defined in the **AsyncEventHandler** class and is moved in the **Handleable** interface. Note that the **AsyncEvent** class of the RTSJ would now be associated with the **Handleable** interface for the parameter of its methods to be passed in, instead of the **AsyncEventHandler** class. The abstract class **Handler** defines **protected** accessor methods for **fireCount**, which were defined in the **AsyncEventHandler** class in the current RTSJ. The **fired()**, **run()**, and **handleAsyncEvent()** methods still remain as abstract. The **SchedulableHandler** class extends the **Handler** class and implements the **Schedulable** interface. This class is still abstract but now provides the default implementations for methods inherited from the **Schedulable** interface, that are related to the feasibility analysis and the accessor methods for scheduling, memory, release, and processing group parameters. Note that the **run()** method is inherited in both the **Handleable** and **Schedulable** interfaces as they both implement **Runnable** interface still remains abstract. Now the **AsyncEventHandler** class extends the **SchedulableHandler** class, providing the implementation-specific AEH algorithm. As the RTSJ defines the **run()** method in the **AsyncEventHandler** class as final and therefore it is final here too. Declaring the **run()** method as final is sufficient for the current RTSJ to shield the AEH algorithm from being modified. For the refactored AEH API it is not sufficient as now the internal hook method which determines the AEH mapping algorithm is open to be overridden. However it is not possible to declare the **fired()** method in the **AsyncEventHandler** as final because the method must be overridden to offer a default AEH algorithm for the **BoundAsyncEventHandler** class (i.e. a static 1:1 mapping). The **fired()** method in the **BoundAsyncEventHandler** class, however, can be declared as final, restricting the **fired()** and **run()** methods of the subclasses not to be modified. Note that, the **fired()** method in the **AsyncEventHandler** class can still be overridden and the programmer must not do so if he/she intends to use the default AEH algorithm provided by the underlying implementation. The **AsyncEventHandler** and **BoundAsyncEventHandler** classes can therefore provide the default implementation of its AEH as the current RTSJ specifies.

In the RTSJ, the interaction between the **AsyncEventHandler** class and the **AsyncEvent** class is not clearly

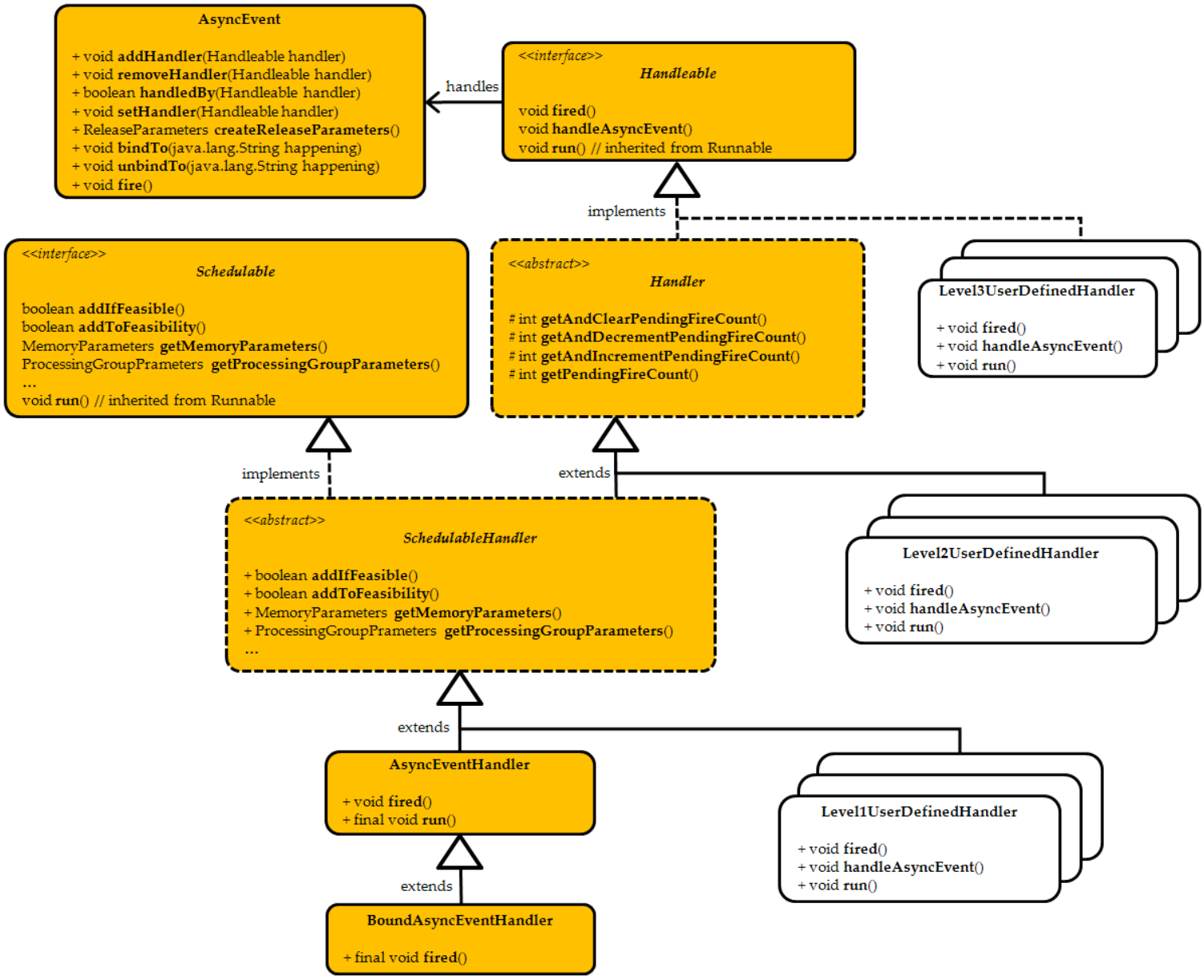

Figure 4. New Refactored Configurable AEH for the RTSJ

defined and hence is implementation-specific in an inconsistent way across different implementations. The RTSJ merely states that [1]:

> When an asynchronous event occurs, its attached handlers (that is, handlers that have been added to the event by the execution of the **addHandler()** method) are released for execution. Every occurrence of an event increments **fireCount** in each attached handler.

Especially the word "*released for execution*" is ambiguous. In OVM, for example, the internal hook method that is invoked when an event is fired is called **releaseHandler** where its dedicated server thread is notified (recall that OVM uses a static 1:1 mapping). Then the server thread in OVM calls the **run()** method of the handler, which repeatedly invokes the **handleAsyncEvent()** method while **fireCount** is greater than zero. jRate, on the other hand, the **fire()** method directly invokes the **run()** method, which in turn calls the **handleAsyncEvent()** method. The mapping algorithm is

performed in the **releaseHandler()** method for OVM and in the **handleAsyncEvent()** method for jRate as a result of this uncertainty. As a consequence the application written for OVM may behave differently in jRate. Therefore here the interaction between the **Handleable** interface (or the **AsyncEventHandler** class) and the **AsyncEvent** is defined to eliminate this ambiguousness such that:

- *When an instance of AsyncEvent occurs (indicated by the **fire** method being called), the **fired** method of instances of the class that have implemented the Handleable interface and have been added to the instance of AsyncEvent by the execution of **addHandler** are explicitly invoked.*
- *The AEH algorithm that governs the execution of handleable objects shall be provided in the **fired** method.*
- *When used as part of the internal mechanism, the **run** method's detailed semantics should follow the below idiom to guarantee that the outstanding fire count should be handled and cleared properly:*

```
public void run() {
   while (fireCount > 0) {
      fireCount--;
      try {
         handleAsyncEvent();
      } catch (...) {
         ...
      }
   }
}
```

The proposed API structure and the definition of the relationship between the **Handleable** interface and the **AsyncEvent** preserve source compatibility with applications written in accordance with the current RTSJ. However, binary compatibility is essentially not supported due to changes made to the **AsyncEvent** class. According to the Java Language Specification [10], changing the name of a method, the type of a formal parameter to a method or constructor, or adding a parameter to or deleting a parameter from a method or constructor declaration creates a method or constructor with a new signature, and has the combined effect of deleting the method or constructor with the old signature and adding a method or constructor with the new signature. If any pre-exsiting binary references a deleted method or constructor from a class, this will break binary compatibility; a **NoSuchMethodError** is thrown when such a reference from a pre-existing binary is linked. In order for the refactored AEH API to preserve binary compatibility with pre-existing binaries, a overloaded version of each method, that are directly related to **AsyncEventHandler** in the **AsyncEvent** class, should be declared, for example for the **addHandler()** method, as follows:

```
// the new addHandler() method
public void addHandler(Handleable handler) {
   ...
}

// the current addHandler() method that calls the new
// addHandler() method after typecasting the parameter
// handler to handleable
public void addHandler(AsyncEventHandler handler) {
   if(handler instanceof Handleable) {
      Handleable handleable = handler;
         addHandler(handleable);
      }
   }
}
```

Using the refactored AEH API, it is now possible for the application programmer to extend the hierarchy in several ways. For example:

- It can provide its own implementation in support of asynchronous event handling. This is done by creating a new application-defined class that extends the **SchedulableHandler** class as illustrated as the **Level1UserDefinedHandler** class in Figure 4. By overriding the **fired()** method, it is now possible to manually create and tune the AEH components, allowing the configuration of the number of servers to be created, the allocation of handlers to servers, and the notification of servers. This comprehensive configurability of AEH offers the following advantages [15]:
  1) handlers in separate servers can be organized so that they do not need to be synchronised,
  2) handlers with tight deadlines can be kept separate from handlers with long execution time,
  3) handlers which do not block can be separated from handlers that block,
  4) heap and no-heap handlers can be bound to separate servers.

  This allows an application to take into account application-specific knowledge about the program for example non-blocking handlers. As an example of an application-defined handler at this level, a more efficient AEH model based on the assumption that handlers do not self-suspend is presented in Section 4.
- It can provide non-schedulable handlers by extending the **Handler** class as illustrated as the **Level2UserDefinedHandler** class in Figure 4. An example application requirement might be to execute an interrupt handler. Here, the **fired()** method may invoke the **run()** method directly rather than by a server thread without entailing unnecessary operations which take place in the standard **AsyncEventHandler** such as queuing and waiting to be scheduled in competition with other activities [4].
- It can provide its own version of the **Handler** class should the need arise. The **Level3UserDefinedHandler** class in Figure 4 illustrates this possibility. **Handleables** at this level can similarly be implemented as **Leve2UserDefinedHandler**. Here, however, the notion of **fireCount** can be bypassed.
- Instances of the class that extends the **AsyncEventHandler** class and overrides the **handleAsyncEvent()**

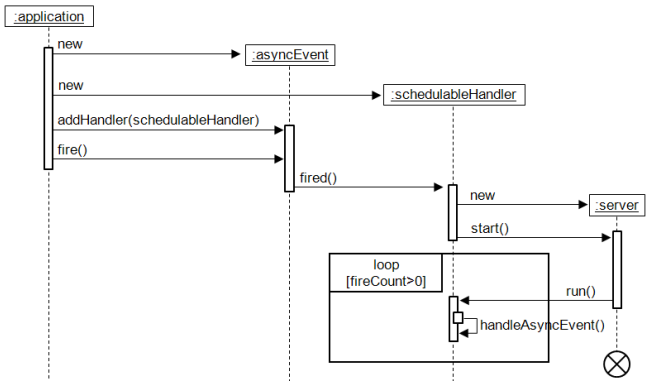

Figure 5. Dynamic-server-creation AEH Model

method will be handled by the default AEH algorithm defined in the **AsyncEventHandler** class unless the programmer overrides the **fired()** method.

Note that as the level of hierarchy is increased (from **Level1** to **Level3**), the extended handleables become lighter and more generalised in the sense that a handleable at **Level1** requires fewer methods to implement than one at **Level2** or **Level3**.

4. Using the Refactored AEH API

We have implemented the refactored configurable AEH in jRate running on top of an open-source RTOS, MarTE OS [12], which allows us to emulate RTSJ-compliant applications on the RTOS using a Linux environment. Using this newly implemented configurable AEH API, various event handling models can be programmed at the application-level. Figure 5 shows the simplest AEH model, that extends the **SchedulableHandler** and is therefore a **Level1UserDefinedHandler**. The AEH model is used in the RI [14] and creates a server every time a handler is released. The entire activities done in the sequence diagram now belong with the application-specific part of the system unlike the previous sequence diagram in Figure 3, that uses the current AEH of the RTSJ. The **fired()** method, that is called when the associated **asyncEvent** is fired, creates and starts a server thread as followings:

```
public void fired() {
     RealtimeThread server = new RealtimeThread(
                this.getSchedulingParameters()){
        public void run() {
           this.run();
        }
     }).start();
}
```

Note that the server thread here only reflects the handler's priority for the sake of simplicity and defines its **run()** method such that it invokes the handler's **run()** method upon creation. Other timing and memory-related parameters can also be reflected on the characteristics of the server thread.

The **run()** method of the handler which is called by the server thread is now written so that the server thread invokes the **handleAsyncEvent()** method or the **run()** method of the handler's logic, that may have been associated with the handler when constructed, repeatedly while **fireCount** of the handler is greater than 0. The code inside the **run()** method of the handler should include the following, which complies with the idiom suggested earlier:

```
do{
   if(this.logic != null)
      this.logic.run();
   else
      this.handleAsyncEvent();
}while(this.getAndDecrementPendingFireCount() > 1);
```

The above code fragment is executed while **fireCount** is greater than 0 and calls the **run()** method of the **logic** that is a runnable object that has been associated with the handler as a parameter when the schedulable handler is constructed or the overridden **handleAsyncEvent()** method, based on the existence of the associated **logic** object. The server thread will be destroyed upon completion as shown in the figure. In the same way as the above dynamic-server-creation model, **Level2** and **Level3UserDefinedHandler** can be constructed by the programmer. Now the handlers at these two levels are not a schedulable any more (i.e. they no longer have parameters related to the schedulable object such as **SchedulingParameters**) and therefore, this could be a good place to design an interrupt handler that is going to be executed directly by the calling thread or the run-time system. The **fired()** method therefore will call the **run()** method directly without creating or notifying a server thread, which will in turn invoke the **handleAsyncEvent()** method.

As presented in Section 2.1, all the RTSJ AEH models have been designed under the assumption that handlers might block. This inhibits the models from taking advantage of non-blocking handlers. This is because non-blocking handlers require a smaller number of servers on average than blocking handlers do: blocking handlers require an additional ready server waiting at the priority of the first handler in the queue to take over the CPU if the currently running handler blocks to prevent priority inversion. In [8] the distinction of the blocking and the non-blocking handler was proposed and a respective AEH model was constructed. The models in the paper were written in the implementation-level and hence cannot be altered or modified by the programmer. However it is now possible to construct the entire AEH models at the application level by programmers using the refactored configurable AEH API. If an application is composed only of non-blocking handlers, the programmer does not have to rely on the standard AEH model that assumes handlers would block and can construct the programmer's own AEH model to execute non-blocking handlers more efficiently. This could be the prime example of the necessity of a configurable AEH API, tailoring the

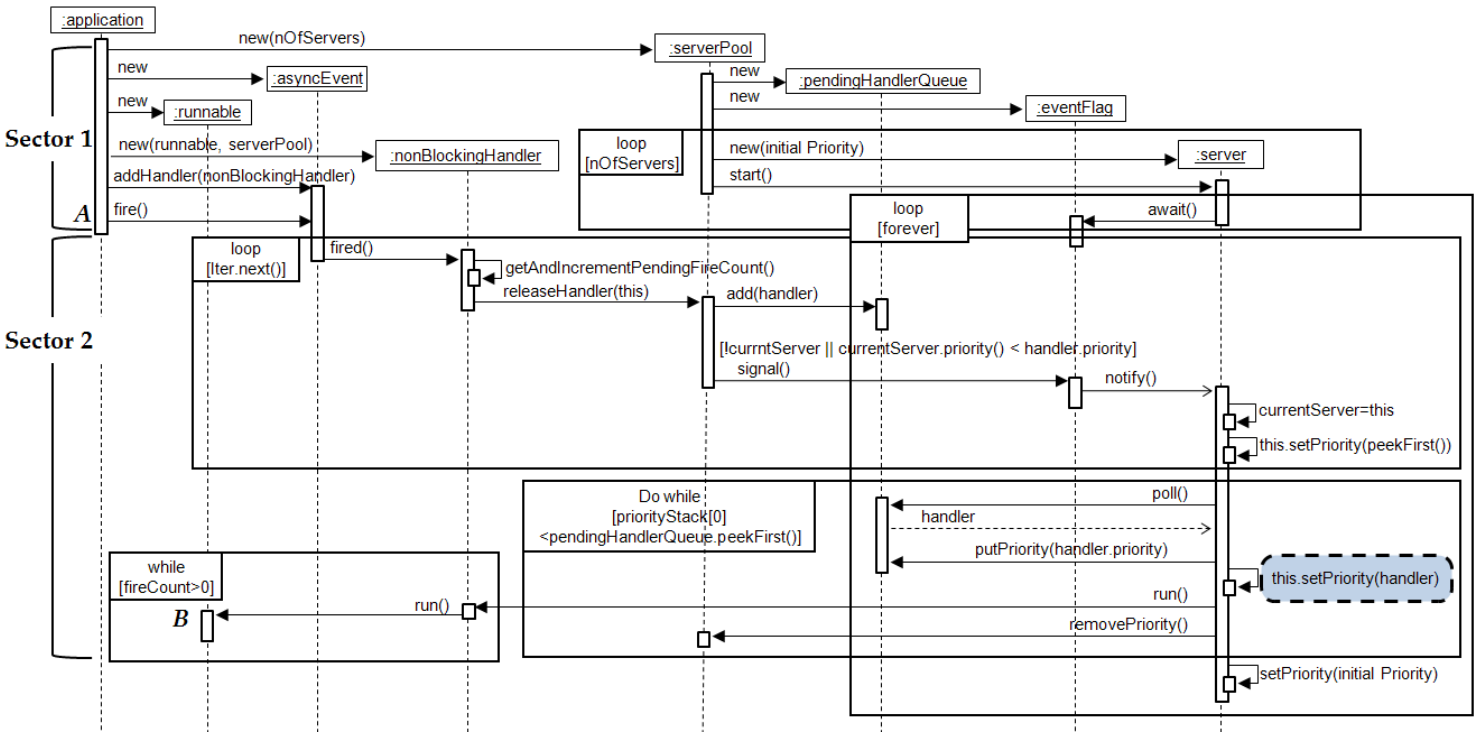

Figure 6. A Sequence Diagram for Non-Blocking AEH model [8]

AEH model to fit their specific needs, the handlers' temporal property of non-blocking in this example.

A sequence diagram of the Non-Blocking AEH model [8] is shown in Figure 6. The model was originally implemented using the current RTSJ AEH API. The two separate sectors in the diagram, **Sector 1** and **Sector 2**, depicted the application-specific and the implementation-specific section of the system, respectively. With the refactored AEH API the entire model can now be constructed at the application-level by the programmer, having full control over the AEH components and the mapping between servers and handlers. The following briefly explains the components and the run-time behavior of the model: Programmers construct a server pool with an integer number, **nOfServers**. The number is then used to specify how many servers to be created in the pool when it is initialized. The pool manages two structural components, **pendingHandlerQueue** and **eventFlag**. The servers in the pool wait on the **eventFlag** when started and they are notified in turn under certain conditions. The queue lists and dispatches the pending handlers in a priority-ordered manner. The **serverPool** is passed to the **nonBlockingHandler** as a parameter when the **handleable** is constructed, forcing that it is only executed by one of the servers in the pool. Creating an **asyncEvent** and attaching the constructed **handleable** to the **asyncEvent** is done in a normal way. And then some time later the event may be fired as shown in the figure. When the pool is constructed, it creates the specified number of servers with an **initial priority**. The servers' initial priority must be equal to or greater than the highest priority of all the **handleables** that servers in the pool will execute to avoid priority inversion. The notified servers will later adjust their priorities to the appropriate values accordingly. Otherwise a handler of a higher priority will be delayed until the server gets a chance to execute and consequently priority inversion will occur. The **fired()** method in the **nonBlockingHandler** is overridden in a way that when an event is fired all the associated handlers are released and the **serverPool** is notified. The **serverPool** is programmed so that it manages the released **handleables** in the **pendingHandlerQueue** and the servers which are notified under this model's notification rules. The invoked server will execute the pending handlers in a priority-ordered manner as long as any concurrency-related side effects are not incurred based on the model's event handling algorithm.

Figure 7 and 8 show the trend of the multiple handlers completion latency which represents the time taken to complete a number of handlers. We performed the test with the three AEH models implemented on the platform, **BoundAsyncEventHandler**, default **AsyncEventHandler**, **NonBlockingAsyncEventHandler**. Numbers of handlers for each model, from 10 to 500, are created and attached to a single event. They all have the identical **Runnable** object to guarantee that their execution time is the same. The priorities

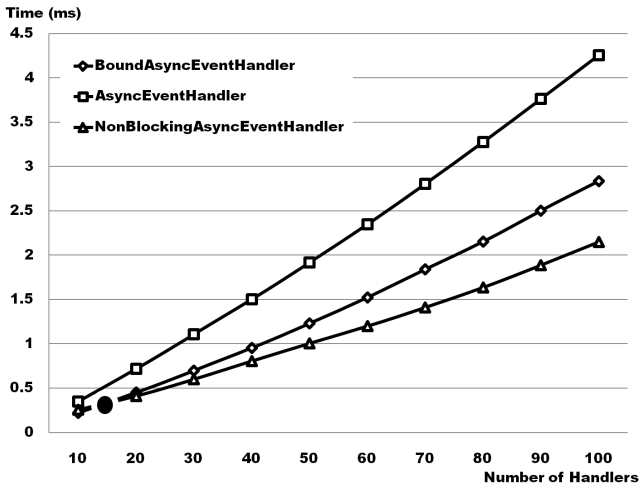

Figure 7. Multiple AEHs Completion Latency Test with 10 to 100 handlers

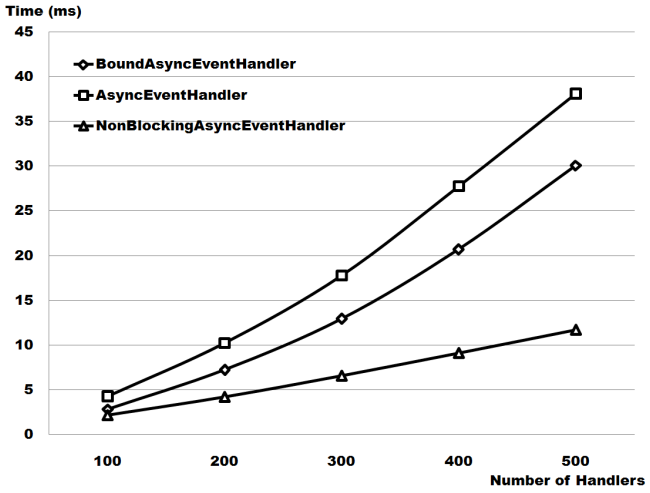

Figure 8. Multiple AEHs Completion Latency Test with 100 to 500 handlers

that are assigned to handlers are equally distributed in the range from the lowest to the highest. When the number of handlers is greater than the priority levels, the same priorities are assigned to handlers. All the handlers used in the test are assumed to be aperiodic (i.e. they do not have a period and a deadline). When the event is fired, all the associated handlers are released for execution. The test measures the time taken from the event being fired to the last handler being completed. To avoid the interference of the garbage collector while performing the test, the real-time thread that fires the **AsyncEvent** and the servers that execute the **AsyncEventHandlers** use scoped memory area as their current memory area.

Figure 7 presents the trend of completion latency for relatively smaller numbers of handlers and Figure 8 demonstrates the trend for larger numbers of handlers. Essentially the latency increases as the number of handlers released increases. The **AsyncEventHandler** model, jRate's standard AEH, takes the longest time to complete the released handlers. The figures also show that the increasing rate of the **BoundAsyncEventHandler** model is greater than **NonBlockingAsyncEventHandler**. At the start, with 10 handlers, the **BoundAsyncEventHandler** incurs the shortest latency. However as the number of handlers increases, it incurs more overhead to complete. Between 10 and 20 handlers, the performance of the **NonBlockingAsyncEventHandler** become better and the performance gap is growing bigger and bigger afterwards. The point where the non-blocking AEH model outperforms the **BoundAsyncEventHandler** model is marked with the black-coloured circle in Figure 7. Figure 8 clearly shows that the performance difference between them becomes more apparent as the number of released handlers increases. When the number of handlers reaches 500, the **NonBlockingAsyncEventHandler** exhibits more than 50% better performance than the other two.

5. Conclusions

This paper has considered the rationale for refactoring asynchronous event handling in the Real-Time Specification for Java. First the AEH models used in some popular RTSJ implementations are examined in terms of their respective run-time behavior and configurability. As the RTSJ does not provide any configurable facilities for AEH, all the implementations examined neither furnish programmers with well-defined documentation for nor offer comprehensive configurability over their AEH models. This fails to instill confidence in programmers that their event handlers will be executed predictably. To address the above issue, a configurable AEH API for the RTSJ is proposed and discussed with respect to its benefits for programmers. The refactored AEH API has been implemented and various AEH models have also been programmed at the application level. The models programmed include the Non-Blocking AEH model that assumes handlers do not self-suspend. We have tested its performance along with the existing AEH models on the platform. The results from the test clearly show the necessity of configurable AEH to allow the application requirements to be tailored. The refactored AEH API lifts the limitations of the current AEH version of the RTSJ, by giving the programmer configurability, as follows:

- **Multiple models for all types of events handlers** – it provides different levels of asynchronous event handling hierarchy, **Level1**, **Level2**, and **Level3UserDefinedHandler**. This allows asynchronous event to be handled in an application-dependent way at the different levels; it is now possible for an application to indicate a

hierarchical implementation strategy for, say, interrupt handlers(**Level2**) or non-blocking handlers(**Level1**).

- **Comprehensive implementation configurability** – the application is now enabled to set the size of the thread pool or to request different handlers be serviced by different pools [8] and this flexible configurability generally provides the following advantages.
  1) application programmers to provide their own AEH model,
  2) the AEH components to be controlled, and
  3) run-time behavior of the components to be regulated.
- **A variety of views of the notion of sporadic handlers** – although the issue has not been directly addressed in this paper, the **fired()** method defined in the **Handleable** interface is where any arrival time constraints are implemented. As this is now under programmer control, user-defined models are possible.
- **Backward compatibility** – an application written in accordance with the current RTSJ can also be executed unchanged in the new system.

The above properties of the new AEH for the RTSJ are significantly beneficial in terms of applications' extensibility and scalability, which can be achieved by specifically tailoring the application to fit their particular needs.

Acknowledgment

The authors gratefully acknowledge discussions with David Holmes and Peter Dibble on the some of the issues presented in this paper.

References

[1] R. Belliardi, B. Brosgol, P. Dibble, D. Holmes, and A. Wellings. Real-Time Specification for Java Ver. 1.0.2, 2008. http://www.rtsj.org/.

[2] A. Corsaro. *Techniques and Patterns for Safe and Efficient Real-Time Middleware*. PhD thesis, Washington University, St. Louis, MO, USA, 2004.

[3] P. Dibble. *Real-Time Java Platform Programming*. Sun Microsystems, USA, 2002.

[4] P. Dibble and A. Wellings. The Real-Time Specification for Java: Current Status and Future Direction. In *7th International Conference on Object-Oriented Real-Time Distributed Computing (ISORC)*, pages 71–77, 2004.

[5] S. Fridtjot. Asynchronous Event Handling in Jamaica from Aicas. Private Communications, April 2008.

[6] D. Holmes. Asynchronous Event Handling in OVM from Purdue University. Private Communications, April 2008.

[7] M. Kim and A. Wellings. Asynchronous Event Handling in the Real-Time Specification for Java. In *JTRES '07: Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems*, pages 3–12, New York, NY, USA, 2007. ACM.

[8] M. Kim and A. Wellings. An Efficient and Predictable Implementation of Asynchronous Event Handling in the RTSJ. In *JTRES '08: Proceedings of the 6th international workshop on Java technologies for real-time and embedded systems*, pages 48–57, New York, NY, USA, 2008. ACM.

[9] D. Masson and S. Midonnet. Rtsj extensions: event manager and feasibility analyzer. In *JTRES '08: Proceedings of the 6th international workshop on Java technologies for real-time and embedded systems*, pages 10–18, New York, NY, USA, 2008. ACM.

[10] S. Microsystems. The Java Language Specification. http://java.sun.com/docs/books/jls/.

[11] F. Parain. Asynchronous Event Handling in Java RTS from Sun Microsystems. Private Communications, March 2007.

[12] M. A. Rivas and M. G. Harbour. MaRTE OS: An Ada Kernel for Real-Time Embedded Applications. *Lecture Notes in Computer Science*, 2043:305–315, 2001.

[13] D. C. Schmidt, M. Kircher, F. Buschmann, and I. Pyarali. Leader/Followers: A Design Pattern for Efficient Multi-Threaded Event Demultiplexing and Dispatching. In *University of Washington*, pages 0–29. Addison-Wesley, 2000.

[14] TimeSys. Real-Time Specification for Java Reference Implementation. www.timesys.com/java.

[15] A. Wellings and A. Burns. Asynchronous Event Handling and Real-Time Threads in the Real-Time Specification for Java. In *Proceedings of the Eighth IEEE Real-Time and Embedded Technology and Applications Symposium*, page 81, Washington, USA, 2002. IEEE Computer Society.

[16] A. Wellings and M. Kim. Processing Group Parameters in the Real-Time Specification for Java. In *JTRES '08: Proceedings of the 6th international workshop on Java technologies for real-time and embedded systems*, pages 3–9, New York, NY, USA, 2008. ACM.

[17] M. Welsh, S. D. Gribble, E. A. Brewer, and D. Culler. A Design Framework for Highly Concurrent Systems. Technical Report UCB/CSD-00-1108, EECS Department, University of California, Berkeley, Aug 2000.