Asynchronous Event Handling and the Real-Time Specification for Java

By

Min Seong Kim
Department of Computer Science
University of York

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

SEPTEMBER 2009

To my Family

Abstract

The Real-Time Specification for Java (RTSJ) was first released in June 2000 due to the needs arisen for safety-critical and real-time systems. The primary objective of the RTSJ was to provide a platform that allows programmers to correctly reason about the temporal behaviour of executing software. The RTSJ has been implemented for both open source and proprietary implementations, providing a foothold for research and deployment in serious applications. As a consequence, its strengths and weaknesses are becoming clear.

One of the areas that requires further elaboration is asynchronous event handling (AEH). The primary goal for asynchronous event handlers in the RTSJ is to have a lightweight concurrency mechanism, on top of predictable and bounded behaviour. This can only be achieved by using a real-time thread to execute as many handlers as possible. Although the RTSJ provides both the flexibility of threads and the efficiency of events, it is generally unclear how to map handlers to real-time threads effectively to achieve the primary motivation, the lightweightness requirement, without causing side-effects (e.g. priority-inversion). Furthermore the RTSJ is silent on how events can be implemented and how their timing requirements can be guaranteed. As a result, most RTSJ implementations simply mapped a released handler to a real-time thread and this results in a 1:1 relationship and therefore undermining the original motivation.

The current RTSJ is also criticised as lacking in configurability as applications were unable to finely tune the AEH components to fit their particular needs. As a result, various handlers with different characteristics must be handled in an implementation-dependent manner and programmers do not have comprehensive configurability over their AEH subsystem.

The main purpose of this thesis is, therefore, first to propose an alternative implementation guideline to guarantee timing requirements of a large number of concurrent asynchronous events in the system while achieving the primary goal of AEH, and second to refactor the current AEH API in the RTSJ to give programmers comprehensive configurability in order for them to be able to specifically tailor AEH components to fit the application's particular needs. By attaining the purpose of this thesis, the development of an efficient (hence scalable) and configurable (hence flexible) RTSJ application using asynchronous event handlers in the real-time Java architecture can be better facilitated.

Contents

List of Figures

List of Tables

Listings

Acknowledgements

The work presented in this thesis would have not been possible without the guidance of my supervisor, Professor Andy Wellings. His continuous support shaped my coarse-grained ideas into this final format. The result is a testimony to his professional and affectionate tutorship. It goes without saying that I am greatly indebted to him for his time, patience and understanding. I would also like to thank my assessor, Dr. Iain Bate, for his great support and invaluable advice for my work.

My deepest gratitude goes to my family for always supporting and believing in me. In particular, I thank my wonderful parents, NakGon Kim and SoJa Cha, and paternal and maternal grandparents, YongTae Kim, DuRae Sung, KyungBek Cha and DukHuep Lee, with all my love. I would also like to thank Aunt SunOk and YueOk, and Uncle YoungDae for their tremendous and endless support. Finally, my sincere appreciation goes to my little sister, NaYoung. Without her support and sacrifice, it would have not been possible to complete this doctorate programme.

York, UK MinSeong Kim
September, 2009

Declaration

I hereby declare that, unless otherwise indicated in the text, the research presented in this thesis is original work, undertaken by myself, MinSeong Kim, between 2005 and 2009 at the University of York, under the supervision of Professor Andy Wellings. I have acknowledged external sources through bibliographic referencing. Certain parts of this thesis have previously been published in the form of a technical report, journal and conference papers. In all cases, the major contributions were made by myself with input and guidance from my supervisor, Andy Wellings.

- Chapter 2 is based on the material previously published as a technical report entitled *Asynchronous Event Handling* (Dept. of Computer Science, University of York MPhil/DPhil Qualifying Dissertation, 2006) [33].
- Chapter 3 is based on work previously published in a conference paper presented at the 5th International Workshop on Java Technologies for Real-Time Embedded Systems (JTRES 07) in October 2007, entitled *Asynchronous Event Handling in the Real-Time Specification for Java* [34].
- Chapter 4 is an extension to work previously published in a conference paper presented at the 6th International Workshop on Java Technologies for Real-Time Embedded Systems (JTRES 08) in October 2008, entitled *An Efficient and Predictable Implementation of Asynchronous Event Handling in the RTSJ* [35]. A revised and extended version of this work is to appear in an international journal, ACM Transactions in Embedded Computing Systems (TECS), later this year [37].
- Chapter 5 is based on work previously published in a conference paper presented at the 7th International Workshop on Java Technologies for Real-Time Embedded Systems (JTRES 09) in September 2009, entitled *Applying Fixed-Priority Preemptive Scheduling with Preemption Threshold to Asynchronous Event Handling in the RTSJ* [36].
- Chapter 6 is based on work previously published in a conference paper presented at the 21st Euromicro Conference on Real-Time Systems (ECRTS 09) in July 2009, entitled *Refactoring Asynchronous Event Handling in the Real-Time Specification for Java* [38].

Chapter 1

Introduction

Real-time and embedded systems are timely reactive systems, meaning that they must react to their surrounding environment within time constraints and at a speed imposed by the environment. The use of such systems in our daily life is rapidly growing and they are widely applied in various application contexts, such as consumer electronics and embedded devices, industrial automation, space shuttle, nuclear power plants, and medical instruments. The number of embedded real-time systems used in a product ranges from one to tens in consumer products and to hundreds in large professional systems. It has been estimated that the market size of embedded systems is 100 times as large as that of desktop and it is projected that the market size of such systems will grow continuously in this decade [18].

Virtually all real-time embedded systems are inherently concurrent and must interact closely with the real world [6]. This inherent concurrency must be efficiently supported. Therefore, providing an efficient concurrency mechanism that facilitates the modeling of parallelism in the real world is one of the main properties of a real-time programming language. Recently there is a trend towards using object-oriented programming languages, such as Java and C++, in real-time embedded systems because they offer several advantages, for example reusability, data accessibility and maintainability, compared to procedural programming languages using well-known object-oriented principles, encapsulation, inheritance, and polymorphism. The use of object-oriented programming also offers a number of additional benefits, including increased flexibility in design and implementation, reduced production cost, and enhanced management of complexity [54].

Java technology with its significant characteristics, such as cost-effective platform-independent environment, relatively familiar linguistic semantics, and support for concurrency, has many advantages for developing real-time embedded systems compared to other object-oriented programming languages. However, due to the non-deterministic behaviour of the concurrency facilities in Java, the use of these facilities often introduces ambiguity for developing real-time embedded systems. The weaknesses that lead to the ambiguity in the Java concurrency model are identified and discussed in [67] as listed below:

- Lack of support for condition variables.
- Poor support for absolute time and time-outs on waiting.
- No preference given to threads continuing after a notify over threads waiting to gain access to the monitor lock for the first time.
- Difficulties in identifying nested monitor calls and thread-safe objects.
- Poor support for priorities.

Although concurrency-related facilities have been reinforced in Java 1.5 by introducing the concurrency utility library, led by Doug Lea in JSR-166 [47], to help alleviate many of the above problems, they have been designed without consideration to temporal guarantees that real-time systems require. Note that, we did not make any specific assumption on concurrency upon which the research work in this thesis is valid.

In order to support a predictable and expressive real-time Java environment, the *Real-Time Specification for Java (RTSJ)* [5] has been developed to provide real-time extensions to Java. The RTSJ has made several modifications to the Java specification to make true real-time processing possible in the context of the standard Java language. For example, threads (or schedulable objects) waiting to acquire a resource must be released in execution eligibility order (i.e. priority) in the RTSJ. This applies to the processor as well as to synchronised blocks. If schedulable objects with the same execution eligibility are possible under the active scheduling policy, such schedulable objects are awakened in FIFO order. This collectively guarantees the order in which threads are granted ownership of a monitor or the order in which threads wake in response to the `notify` or `notifyAll` method. The primary goal of the RTSJ is, therefore, to provide a platform, a Java execution environment and application program interface (API), that lets programmers correctly reason about the temporal behaviour of executing software. To achieve this goal, the RTSJ focuses on strengthening the Java concurrency model and enhancing the standard Java language in the following seven areas:

- Thread Scheduling and Dispatching
- Memory Management
- Synchronisation and Resource Sharing
- Asynchronous Event Handling
- Asynchronous Transfer of Control
- Asynchronous Thread Termination
- Physical Memory Access

Since the RTSJ has been approved by the Sun Java community, it has become one of the most promising object-oriented programming languages for real-time systems in recent years. As the RTSJ has been implemented, formed the basis for research and used in serious applications, some strengths and weaknesses are emerging. One of the areas which require further elaboration is *asynchronous event handling (AEH)*. This is due to the fact that the RTSJ neither provides any implementation guidelines (in order to allow variation on implementation decisions [5]) nor facilitates a configurable framework that allows the programmer to finely tune the AEH subsystem to fit the application's particular needs [67]. As mature implementations of the RTSJ are becoming available, an assessment of the current AEH techniques used in RTSJ implementations is, therefore, timely.

1.1 Motivation

In an attempt to provide the flexibility of threads and the efficiency of event handling, the RTSJ generalises Java's mechanism for asynchronous event handling by distinguishing between events, something that can happen, and event handlers, logic that will be scheduled for execution when the events happen. This distinction in the RTSJ is realised by introducing the notion of asynchronous events via the `AsyncEvent` class, and their associated handlers via the `AsyncEventHandler` class. Asynchronous events in the RTSJ 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 `AsyncEventHandler` objects, and a single `AsyncEventHandler` can be bound to one or more `AsyncEvent` objects). Also handlers can be associated with POSIX signals for the case where the RT-JVM is implemented on top of a POSIX-compliant operating system.

RTSJ defines preemptive priority-based scheduling and two types of schedulable objects (i.e. instances of classes that implement the `Schedulable` interface): `AsyncEventHandler` and `RealtimeThread`. From the application programmer's perspective, they behave in the same manner. In practice, however, real-time threads provide the vehicles for execution of `AsyncEventHandlers` [69]. In other words, an `AsyncEventHandler` will be executed some point later by an implementation-defined real-time thread[1] some time in the future. Therefore it is necessary for an implementation to bind a handler to a server thread and this binding is typically performed at run-time dynamically. The time at which a handler is bound to a server thread and the number of such server threads required form the main distinguishing characteristics between AEH implementations. If this binding latency, which is inevitably incurred when two objects are being attached to each other should be avoided, a `BoundAsyncEventHandler` can be used to eliminate it. Once a `BoundAsyncEventHandler` is bound to a real-time thread, the thread becomes the only vehicle for the execution of that particular handler, establishing a static 1:1 relationship.

The driving design goal of the AEH model of the RTSJ is to have *a lightweight concurrency mechanism* [5] that is capable of supporting hundreds (if not thousands) of handlers; consequently handlers must not incur the same overheads as real-time threads [5, 16, 69]. Therefore, if a system needs many `AsyncEventHandlers` and the binding latency is not critical, the use of `BoundAsyncEventHandler` should be minimised as they incur the same overhead as real-time threads. The lightweightness can only be achieved by using as few as possible server threads to execute as many as possible handlers. Doing this, results in smaller data structures in the real-time virtual machine, less stack space for server threads, fewer context switches between server threads etc. The trade-off is an increase in the latency between the event being fired and a handler being assigned to a server thread for execution, compared to the static 1:1 relationship used for bound asynchronous event handlers.

The RTSJ however does not provide any guidelines on how these events and their han-

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

dlers can be implemented and how their timing requirements can be guaranteed [69] to achieve this lightweightness, allowing variation in implementation decisions [5]. Regardless of how well the implementation is designed, as the number of threads in a system grows, operating system overhead increases, leading to a decrease in the overall performance of the system. There is typically a maximum number of threads that a given system can support, beyond which performance degradation occurs [72]. Therefore, the key challenge in implementing asynchronous event handling for the RTSJ is to limit the number of server threads without jeopardising the schedulability of the overall system. A set of tasks said to be schedulable if there exists at least one algorithm that can produce a schedule where all tasks (handlers in the context of this thesis) in the set can be completed according to a set of specified constraints (i.e. deadlines). By limiting the number of concurrent server threads for execution of released handlers and still maintaining their schedulability, the system can avoid throughput degradation, and the overall performance will be higher than the unconstrained thread-per-handler or run-time thread creation models. This, therefore, will enable the system to be more *efficient* and hence *scalable*. Here, an AEH algorithm is said to be efficient if the algorithm can execute a larger number of released handlers using a smaller number of server threads. Likewise, an AEH algorithm is said to be the most efficient if it can execute a certain number of handlers using the least number of server threads. Note that server threads themselves should be lightweight enough as fewer server threads with much higher overheads do not make AEH more lightweight[2]. Consequently efficient AEH algorithms have more space for additional server threads without performance degradation, resulting in having more scalability.

In many RTSJ implementations, however, a released handler is simply executed in the context of a currently available server or the run-time creates an *ad-hoc* server dynamically for the execution of the released handler [12, 29, 60]. In [16, 69], possible implementation strategies for AEH in the RTSJ and some of their corresponding schedulability analyses have been proposed. One of the implementation strategies that appeared in both papers is to use a priority-ordered queue (or one queue per priority level) and multiple servers. This approach is more flexible than a single server approach in terms of dealing with handlers whose execution might be suspended as a result of issuing a wait or a sleep method call in the `Thread` class – for example, by calling the `Thread.sleep`, `Object.wait`, or `Thread.join` methods, or any blocking I/O call. Hence, a server thread executing such a

[2]Later in Chapter 4 we have demonstrated the overhead of a handler is lightweight enough not to make the overall performance of AEH worse.

handler may not be able to complete its execution without suspending. Once suspended, it is unable to execute any other handler. These handlers are called *blocking (or self-suspending)*[3] handlers in this thesis. The implementation challenge in meeting these goals is made more difficult because the RTSJ is required to create a server, when a handler is released and all the servers in the system are blocked, in order to avoid unbounded priority inversion. The dynamic creation however is relatively expensive in terms of both time and memory. Also there are other implementation issues to address, such as defining the points where and when a server should change its priority to reflect the priority of the handler which the server is executing, and clarifying the circumstances under which a server is allowed to continue executing another handler after completing the execution of the current one. Without having these issues addressed, it is likely that the model causes unnecessary context switches or leads to invocation of superfluous servers.

Also the current AEH API in the RTSJ is criticised as lacking in configurability [38, 46, 69] as it does not provide any means for programmers to have fine control over the AEH algorithm and its components, such as the mapping between server threads and handlers. The `AsyncEventHandler` class provides a default and immutable AEH implementation which mandates the way of handling asynchronous events. All asynchronous events must be handled in an implementation-dependent way defined by the default AEH implementation. Consequently, the programmer cannot tailor the AEH subsystem to fit the application's particular needs. For example a handler might be used to function as interrupt handlers. Such handlers should normally be executed directly by the calling thread or by the JVM at the highest priority level, and they do not have to be schedulable objects which incur the associated overheads such as mapping and competing with other schedulables for the CPU. In order to provide configurability for AEH, the current AEH API must be refactored to give programmers options to execute handlers with different characteristics in a different way. Furthermore, the RTSJ does not require the documentation of the AEH implementation. These issues collectively affects the predictability of an application; it is difficult for programmers to project how their handlers in the application will behave at run-time and to regulate their behaviour. Therefore the current AEH API

[3]We do not see the potential delay incurred when accessing a synchronised statement or method as a block. This is because priority inheritance will cause another handler already bound to a server to execute. For example, if the server dedicated to a priority level blocks, the remaining pending handlers (if any) at the priority level will also block until the server unblocks. In the mean time, a server associated with a lower priority level will run, as a result of priority inheritance. Hence, the processor will not necessarily become idle.

in the RTSJ imposes 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 handlers with different characteristics such as interrupt or non-blocking handlers.

- The lack of implementation configurability – the application is unable to configure its AEH components (e.g. the size of the thread pool or to request different handlers be serviced by different pools).

The underlying idea behind this thesis, therefore, is first that an AEH subsystem that minimise the number of server threads for execution of released handlers is better able to cope with systems that have a large number of handlers and hence is more scalable, and second that offering comprehensive configurability over its AEH components is beneficial for the programmer as they can be specifically tailored to fit the application's particular needs. In order for the RTSJ to become the first concurrent object-oriented real-time programming language widely accepted[4], it is therefore necessary to address the above issues.

1.2 Objectives and Contributions

This thesis mainly concerns the efficient implementation and the comprehensive configurability of AEH in the RTSJ, which offers greater flexibility and better efficiency to encompass handlers with different characteristics, while using the least possible number of server threads, but without jeopardising the overall schedulability of the system. The work in this thesis is therefore based on the premise that:

- The primary goal of AEH in the RTSJ is to have a lightweight concurrency mechanism that is capable of support hundreds (if not thousands) of handlers. However, the RTSJ does not provide guidelines on how asynchronous event handlers can be mapped onto server threads and how their timing requirements can be guaranteed to achieve the lightweightness requirement.

- The RTSJ does not provide any means for programmers to have fine control over AEH such as the mapping algorithm and its components. As the result, all asynchronous events must be handled in an implementation-dependent way.

[4]Currently, C and C++ are widely accepted and used in real-time systems.

In the absence of the AEH implementation guidelines and the lack of configurability of RTSJ AEH, the scalability of a RTSJ implementation to scale to systems of hundreds of events is practically inhibited as most of the RTSJ implementations take a 1:1 AEH mapping approach, where an invoked server thread executes only one handler and goes back to the waiting state where it sleeps until invoked again[5]. Therefore, the major aim of the research is twofold:

- How to provide a well-defined guideline to achieve the primary goal of AEH in the RTSJ, the lightweightness, and
- How to facilitate comprehensive configurability over AEH components so that the AEH subsystem can specifically be tailored to fit the application's particular needs.

The following two propositions synthesise the central research hypothesis:
The current version of the RTSJ does not facilitate either a lightweight concurrency mechanism or comprehensive configurability over its AEH. The efficiency and flexibility of AEH in the RTSJ can be better achieved if the implementation

Proposition I: is aware of whether asynchronous event handlers are

- Blocking or Non-blocking, and
- Periodic (Hard) or Non-periodic (Soft).

to be able to reduce overheads both in time and space, and still meet the application's timing requirements.

Proposition II: facilitates comprehensive configurability over its AEH facilities as the lack of the property inhibits the scalability of the RTSJ implementation from scaling to systems of hundreds of events. The problem is exacerbated by the presence of asynchronous event handlers with different characteristics presented in Proposition I.

In order to prove the above propositions the following objectives are set out and discussed in detail in subsequent chapters:

Objective 1 Investigating and implementing AEH mapping algorithms that use the minimum number of server threads,

Objective 2 Drawing the distinction between blocking and non-blocking handlers to further reduce the required number of server threads,

[5]For more information about the AEH mapping models, see Section 3.1.

Objective 3 Separating non-aperiodic and aperiodic handlers thereby allowing the use of preemption threshold technique for hard (or periodic) handlers: this technique lets the AEH subsystem further reduce the number of server threads for execution of released handlers even in the worst case.

Objective 4 Refactoring of the current AEH API in the RTSJ, which results in giving programmers full configurability over AEH subsystem,

Meeting these objectives forms the main contributions of this thesis. Therefore, the research work of this thesis is based on the above problems and observations. Achieving these goals will facilitate the development of an efficient and configurable RTSJ application using asynchronous event handlers in the real-time Java architecture. The rationale behind the research is mainly based on defining a more efficient AEH algorithm and refactoring the current AEH API in the RTSJ to give programmers comprehensive configurability. Note that, in the RTSJ, the semantics for the base scheduler essentially assume a uniprocessor execution environment. While implementations of the RTSJ are not precluded from supporting multiprocessor execution environments, no explicit consideration for such environments has been given in the specification. Therefore a uniprocessor execution environment will also be the main focus of the research work for this thesis.

1.3 Organisation of the Thesis

In accordance with the motivations and objectives of the research, the thesis will be organised with eight chapters. Brief descriptions of the remaining chapters are given below:

Chapter 2. Event Handling in Real-Time Systems

This chapter explores concurrent programming models, thread and event-based, emphasising the flexibility of threads and efficiency of events. Also event handling facilities provided by general-purpose and real-time systems, including programming languages, middleware systems, and operating systems are discussed. Asynchronous event handling in the RTSJ is also closely investigated, with particular emphasis to the API relevant to schedulable objects, and its possible implementation strategies.

Chapter 3. Appraising AEH Models in Current RTSJ Implementations

This chapter examines various asynchronous event handling algorithms in some popular RTSJ implementations with respect to efficiency and flexibility. To evaluate asynchronous event handling implementation of two non-trivial RTSJ implementations, jRate [12] and Java RTS [50], a model checking tool, UPPAAL [61], has been extensively used. This chapter identifies the multiple-server switching phenomenon (MSSP) [37] found in the Java RTS AEH. It takes place whenever the current running server changes its priority to that of the handler to execute, due to queue replacement policy in that most real-time OSs and middleware systems put a thread that has changed its priority at the tail of the relevant queue for its new priority. The MSSP can be found in any system where the mapping between executioner and executionee is required and the priority of executioners dynamically changes to execute the larger number of executionees. Java RTS is an example system which suffers from the phenomenon.

Chapter 4. Efficient AEH for the RTSJ

In this chapter, more efficient models for asynchronous event handling are proposed and implemented. The underlying idea beneath the proposed models is to use as few servers as possible. This is to achieve the primary intention of AEH in the RTSJ. They are designed to serve handlers with different characteristics, blocking and non-blocking. To validate the idea, the dynamic 1:N mapping model is discussed and emphasised for its efficiency and scalability. Based on the dynamic 1:N mapping model, the notion of *critical sequences*[6] for blocking and non-blocking handlers is defined. A critical sequence states the handler releasing scenario under which the least upper bound of server threads is required. For a set of blocking handlers, the critical sequence occurs when a handler is released after or at the release time, and before or during self-suspension of a previously released handler. For a set of non-blocking handlers, the critical sequence occurs when handlers are released in a priority-ascending manner in turn during the previously released handler being executed. Therefore, each blocking handler is required to have a single server as it is generally not possible to know when a handler will self-suspend, and non-blocking handlers are collectively required to have the number of servers as the number of priority levels. Based on this, the equations that calculate the least upper bound of servers required for both blocking and non-blocking handlers are derived for a given set

[6]This differs from *critical path* that is the sequence of tasks with dependencies which add up to the longest overall duration.

of handlers. The blocking and non-blocking AEH implementations are constructed, based upon the dynamic 1:N mapping model, which also utilise the notion of critical sequences. The critical sequences for blocking and non blocking handlers are particularly important as they offer the ground upon which the least upper bound of server threads can be derived and this allows programmers to safely assume that all released handlers can all be handled with a proper set of the particular server notification and relinquishment rules.

Chapter 5. Applying Fixed-Priority Preemptive Scheduling with Preemption Threshold to AEH in the RTSJ

This chapter discusses fixed-priority preemptive scheduling with preemption threshold (FPPT) which reduces the number of effective priority levels for a given task set. To perform the schedulablility analysis for the FPPT, tasks under consideration should be periodic. Therefore, periodic and non-blocking task sets can successfully be scheduled with the FPPT as the least upper bound of server threads for non-blocking handlers depends on the number of priority levels. In this chapter, the non-blocking AEH implementation is extended to support FPPT to reduce the least upper bound of server threads required even at the critical sequence which completely depends on the number of priority levels. Applying this technique to AEH in the RTSJ further reduces the number of server threads required for periodic and non-blocking handlers, better achieving the primary goal of AEH, the lightweightness.

Chapter 6. Refactoring AEH in the RTSJ

This chapter presents the refactored AEH API for the RTSJ. As the RTSJ does not provide any configurable facilities for AEH, most RTSJ implementations examined in this thesis neither furnish programmers with well-defined documentation nor offer comprehensive configurability over their AEH implementations. This fails to instill confidence in programmers that their event handlers will behave predictably at run-time. To address this issue, the configurable AEH API for the RTSJ is proposed and discussed with respect to its benefits for programmers. The refactored AEH API lifts the limitations of the current AEH API in the RTSJ, by giving programmers comprehensive configurability. This enables AEH components to be specifically tailored to fit the application's particular needs.

Chapter 7. Conclusions and Future Work

The final comments about the research results are given in this chapter. Additionally, some directions for further research are also presented.

Chapter 2

Event Handling in Real-Time Systems

A major application area of concurrent programming is real-time systems [7, 67] as they are inherently concurrent due to the parallelism that exists in the real-world objects which they are monitoring and controlling. A primary issue associated with the production of software for real-time applications that exhibit concurrency is how to express that concurrency in the structure of the program.

> *Concurrent programming is the name given to programming notation and techniques for expressing (potential) parallelism and solving the resulting synchronisation and communication problems. Implementation of parallelism is a topic in computer systems (hardware and software) that are essentially independent of concurrent programming. Concurrent programming is important because it provides an abstract setting in which to study parallelism without getting bogged down in the implementation details [6].*

Until recently, general-purpose computer systems were centralised, isolated, and expensive. Today, they are parallel, distributed, networked, and inexpensive. However, advances in software construction have failed to keep pace with advances in hardware [58]. To some extent, this is a result of the fact that current programming languages were conceived for sequential and centralised programming. Programs written in procedural languages such as Pascal, C, FORTRAN and COBOL have a single thread of control. Generally, they have only one path at any particular execution of the program except when input data varies, in which case the path through the program may differ. Given the fact that the inherently concurrent nature of real-time and embedded systems, the software must control the simultaneous operations of the coexisting hardware or software components in the surrounding environment. As a consequence, providing an easy and efficient framework for expressing parallelism in a real-time application has been an important research area in the real-time community. Programming languages intended for use in the domain of the real-time system therefore should have greater expressive power to provide the programmer with primitives that match the application's parallelism. On top of this, there are two other fundamental motivations for the support of the concurrent programming model [67]:

- To fully utilise the processor
- To allow more than one processor to solve a problem

To cater for the necessity of the concurrency, several modern high-level programming languages, such as Ada, C#, and Java, have direct support for concurrent programming with the notion of threads and events.

In this chapter, general event handling techniques and facilities in high-level programming languages, middleware systems, and operating systems are presented to closely investigate how they are implemented in terms of concurrent programming models to achieve efficiency and flexibility. This chapter therefore first explores programming models for concurrency, thread and event-based, emphasising the flexibility of threads and the efficiency of events in Section 2.1. Mixed programming, that is a hybrid form of thread and event-based programming that can be used to build more scalable and flexible systems, is discussed next. (Asynchronous) event handling facilities provided by general-purpose and real-time systems, including programming languages, middleware systems, and operating systems are presented in Section 2.2. Each framework provides a slightly different scheme towards its event handling, and the difference in the mapping between executionee (i.e. handlers) and executioner (i.e. threads) is investigated in detail. Section 2.3 provides an introduction to asynchronous event handling in the RTSJ with particular emphasis to the API relevant to schedulable objects. The section also discusses AEH implementation issues with respect to efficiency and flexibility for the RTSJ.

2.1 Concurrent Programming Models

Building highly concurrent systems, such as large-scale real-time systems managing many information flows at once and maintaining peak throughput when demand exceeds resource availability, is inherently difficult [72]. Also such systems impose a number of new challenges to application designers [64] (e.g. the demand of scalable data structures, maximum resource utilisation, race conditions, etc.). Many approaches have been proposed to build concurrent systems, which generally fall into the two categories of *thread-based* and *event-based*. The debate over whether threads or events are best suited for modelling concurrency has been furious for decades [21]. For a given set of tasks, a scheduling algorithm is said to be feasible if the tasks can be scheduled so that no missed deadlines ever occur. The key issue of this debate is whether thread-based or event-based programming is better suited for achieving *feasible scheduling* of concurrent activities in real-time systems (or parallel programming techniques to provide better performance of the application in the case of general-purpose systems).

The thread-based programming model has become the dominant form of expressing concurrency. Thread support is standardised across most operating systems, and is well-established. It is incorporated in modern languages, such as Java and Ada. While threads are a commonly used concept for expressing concurrency in many high-level languages, the characteristics of thread implementations has led many developers to prefer an event-based approach due to its high resource usage and scalability limits [72]. However, event-driven systems are generally built from scratch, and depend on mechanisms that are not well-supported by most languages and operating systems. Furthermore, using the event-based programming model for concurrency can be more complex to develop and debug than its counterpart [64].

There also has been another long debate among programmers, language designers and operating system designers associated with providing support for concurrency as to whether it is appropriate to provide the support in a language or whether this should be provided by the operating system only. Arguments in favour of supporting concurrency in the programming languages include the following [6, 7]:

1. It leads to more readable and maintainable programs.
2. Compilers for languages that are unaware of potential concurrent executions of program components may introduce race conditions in an attempt to optimise code

execution.

3. There are many different types of operating system, defining the concurrency in the language makes the program more portable.
4. An embedded computer may not have any resident operating system available.

Arguments against concurrency in a language include the following:

1. Different languages have different models of concurrency; it is easier to compose programs from different languages if they all use the same operating system model of concurrency.
2. It may be difficult to implement a language's model of concurrency efficiently on top of an operating system's model.
3. International operating system and middleware standards, such as POSIX and the .NET Framework, have emerged, and therefore programs become more portable.

This thesis, however, assumes that the approach of supporting concurrency in the programming language level is more up to date and suited to cater for the current trend of high level programing languages such as Ada, C#, and Java. In this section, the two major concurrent programming models are first examined. Based on this, the mixed-mode programming model is then discussed, emphasising its flexibility and efficiency that cannot be accomplished with only providing a single abstraction of concurrency, thread or event.

2.1.1 Thread-based Programming

Interrupt (or event) handlers had originally been used for notifying the application of the occurrences of both external and internal events and dealing with non-periodic I/O activities. However, the thread abstraction was introduced to alleviate the programming difficulties of using interrupt handlers (i.e. blocking is not permitted and long running interrupts unnecessarily delay other tasks as interrupt handlers are typically performed by the system with the highest priority). Threads are more flexible and can block. Under this programming model, the functionality of the concurrent activities (e.g. controllers) in the system is represented as a thread. They can be executed either based on the system clock (time-triggered), or on the events fired (event-triggered) in the environment. In a time-triggered system, the controller can be invoked either periodically or sporadically [45]. This approach models the recurring behaviour of the physical system controlled by the real-time application. It monitors the environment in order to determine 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 target object. For example, in an environment where sensor readings are used to control actuators, application threads will be triggered periodically to guarantee that sensors are polled in time. Other threads might have more irregular time triggers (e.g. *at most* or *at least* once per interval). In an event-triggered system, sensors in the environment are activated when the real world object enters into anticipated states. Then the events are signalled to the controller via interrupts. In the above example, the sensors might send an interrupt signal to the real-time system every time they perceive new readings informing the system that events that need to be handled have arrived.

One of the main advantages of the thread-based programming model is that it is widely accepted by the real-time community as a *natural* and *powerful* way of expressing concurrency in large scale applications, especially high concurrency I/O servers [64]. The concept of threads enable parallel programming notion to be easily implemented, providing high performance, particularly for multiprocessor systems [49]. This high performance can be further excelled by relatively better support from advanced facilities such as OS, libraries, debuggers and languages. Additionally, thread-based systems can achieve improved safety and performance with only minor modifications to existing compilers and run-time systems [64], which is an extremely powerful concept for system design. The system designer can often be given a choice whether to implement the control algorithm as time-triggered or event-triggered [67] under the thread-based programming model; generally event-triggered implementation is considered to be often more flexible, mostly used in the non-safety critical community where as time-triggered one to be more predictable, prevailed in safety critical real-time applications [39].

One of the most well-known difficulties imposed by the thread-based programming model is synchronisation paradigms which are either too low-level, or too coarse-grained [62]. Their error-prone low-level mechanisms such as a semaphore and coarse-grained mechanisms such as a monitor do not allow for much parallelism by overspecifying the amount of synchronisation required. Furthermore, there are situations where the thread-based programming model is not appropriate including the following [49, 62] cited in [67].

- the external objects are many and their control algorithms are simple and non-

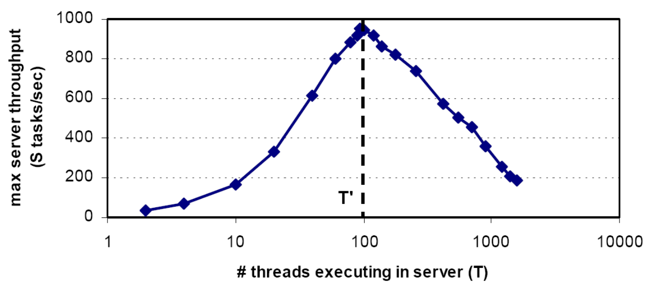


Figure 2.1: Performance versus Number of Threads

blocking, and

- the external objects are interrelated, and their collective control requires significant communication and synchronisation between the controllers.

In the former case, 1 to 1 mapping between a thread and an object controller may lead to a proliferation of threads along with per thread overhead [67], which will likely reduce the overall performance of the system as managing a large number of concurrently active threads imposes larger latency, context switching, cache misses, release jitter, and complexities of implementation. This is because there is typically a maximum number of threads T′ that a given system can support, beyond which performance degradation occurs. This phenomenon is demonstrated clearly in Figure 2.1 [72]. In the figure, it is shown that as the number of concurrent threads T increases throughput increases until T = T′, after which the throughput of the system degrades substantially.

In the latter case, communication and synchronisation protocols inevitably required for the safe use of shared data are extremely complex [62] and this introduces all the well-known pitfalls of using threads such as blocking, priority inversion and deadlock regardless of how well the application is designed.

2.1.2 Event-based Programming

The high resource usage, scalability limits and performance degradation of the thread-based programming model [64, 72] have led many programmers to prefer the event-based counterpart. Under this paradigm [6], typically each event has an associated handler. When events occur, they are queued and one or more server threads take events from the queue and serve their associated handlers. When a server has finished executing a handler, it takes another event from the queue, executes the handler. This iteration of execution lasts until the queue is exhausted. The order of the event queue in which events are placed upon firing can be vary, depending on the queue placement policy such as FIFO (First-In-First-Out), priority-ordered (if the language supports the notion), deadline-ordered (if the underlying platform allows deadline monitoring), etc. To enable effective schedulability analysis for real-time systems, the event queue should be priority-ordered or deadline-ordered. With this approach, fewer threads (one or more servers) is required in comparison with the thread-based approach, reducing the overall memory usage and alleviating the performance degradation issue, presented in Figure 2.1.

The event-based model can be characterised by the separation of events and the handling of events. An event can be generated by sensor interactions with the physical environment such as temperature exceeding a threshold or changes made in the surroundings. It could also be internal to a system such as reading a certain sequence of bytes in a data stream (e.g. a particular tag in a text file). Another fundamental difference between the thread- and the event-based programming model is that in the case of event-based programming the application typically controls itself (i.e. without much help from the underlying operating system), while in the case of the thread-based programming model additional features of the operating system or language primitives should be available to apply safe synchronisation and communication among threads, otherwise programmers have to provide their own synchronisation and communication protocols, which can be complex and error-prone.

In recent years, the event-based approach has been widely considered as the best way to write highly concurrent applications [6, 21, 49]. This is mainly due to its advantages that address the issues associated with the thread-based programming model including the following:

- Complexities of concurrency: It spares the complexities of concurrency [62] by eliminating complex and expensive synchronisation and communication protocols required for the thread-based programming if only one server thread is used; otherwise the model unnecessarily forces the programmer to cope with latent data races, priority inversion and deadlock.
- Error debugging: The convenience of using threads is not worth the errors they encourage [64, 72], except for multiprocessor systems. Since systems would become exponentially complex in proportion to its size, the synchronisation and communication errors, particularly potential deadlocks when the system is composed of many

modules are extremely hard to debug.

- Complexities of implementation: Event handlers in the event-based programming which are normally short-lived [49] and consist of relatively simple logic [6] make the application design and implementation more manageable to maintain and extend.

- Performance: The throughput of a simple server using kernel-supported threads degrades rapidly as many concurrent threads are required [21, 72], which is undesirable property, especially in real-time applications as they must gracefully handle overload conditions at any point while running.

- Memory usage: The event-based approach introduces a lower overhead as fewer stacks are required [49] as it only need allocate the relatively small amount of memory, not a whole thread stack [21] - this reduces overall memory usage.

The event-based model alleviates all the above critical problems associated with the thread-based programming model. If the mapping of event handler to server thread is under program control, it is possible to ensure there is no need for explicit communication between handlers [6].

One of the most notable disadvantages of controlling all external objects by event handlers is that it is difficult to have tight deadlines associated with event handlers. This is because long-running (or blocking) handlers make application non-responsive, and may even cause the handlers to miss their deadlines[1]. Also, the event-based approach cannot fully take advantage of multiprocessor systems for performance [72] unless multiple server threads are used. Furthermore, it is even difficult to extend the model on such systems as the handlers typically assume that there is no contention for shared resources [6]. Although, the event-based programming does not suffer from unbounded priority inversion by implementing a priority inheritance protocol, there still is a certain non-preemptible time. If a single thread is used to serve all the event handlers of the system, the server thread is essentially a shared resource for which event handlers are competing. Therefore, a low priority event handler, that is currently running, may delay the execution of a higher priority handler until it completes. If the low priority handler blocks (e.g. waiting for

[1]The consequence of a deadline miss will depend on whether the real-time system is soft or hard. A real-time system is said to be soft if meeting its deadline is desirable for performance reasons, but missing its deadline does not cause serious damage to the environment and does not jeopardise correct system behaviour. On the other hand, a hard real-time system may cause catastrophic consequences on the environment under control if it misses its deadline.

I/O) the situation is even worse as the whole system will halt until the handler regain the execution. In either case, the real-time application will suffer from the priority inversion (although bounded if multiple server threads and a priority inheritance algorithm are supported) and this clearly affects the application's predictability and responsiveness. If multiple server threads are used to handle events to alleviate the problem depicted above, concurrency becomes an issue again as handlers should now assume that there may be some contention for shared software resources between server threads [6]. The single and multiple server approach to model the event-based programming model are revisited later in Section 2.3.2.

2.1.3 Mixed Programming

In many real-time control applications, periodic activities (e.g. sensory data acquisition, system monitoring, etc.) represent the major computational demand in the system [8] and such activities need to be cyclically executed at specific rates within predetermined deadlines, which can be derived from the application's temporal requirements [6]. These cyclic activities that generally form the main part of the application logic and are executed throughout their lifetime are appropriate to be implemented using time-triggered threads. On the other hand, the applications have to respond to certain unanticipated events which occur at random instants of time from both the internal and external environments to model parallelism with the real world [45]. These additional control logics often arrive aperiodically or sporadically within known intervals. This should be done in parallel with the main application functionality, forcing the application to handle them as they occur, outside its main control algorithm. This instance in general illustrates why it is not sufficient to support only one abstraction to provide the programmer with flexible and efficient control over the possible activities in the system. By focusing on a single abstraction for concurrency, thread or event, traditional models for real-time and embedded software may ignore a great deal of the richness and complexity that is present in actual systems.

Comparison between the event-based and the thread-based programming model in terms of structure, performance, logical soundness, elegance, and correctness is conducted in [30], the duality of each model. It concludes that results which traditionally illustrate deficiency or sufficiency of one programming model compared to the other are not borne out by the natural supremacy or fundamental properties, but an artefact of the implementations [64] and supporting facilities [72] (e.g. compilers). Also the decision regarding which paradigm to use is application-specific [30] as different applications have different requirements.

The event-based approach is useful for obtaining high concurrency [49], but when building actual systems, threads are necessary in many cases for exploiting multi-tasking parallelism and dealing with blocking I/O mechanisms [72]. The two approaches of concurrent programming actually represent each end of extreme cases for designing concurrency. The design space for concurrency is not limited to these two approaches. Rather, there is an optimal configuration between these two extremes according to the needs of the system. For example, by limiting the number of concurrent threads running in the system and buffers incoming tasks in a queue from which the threads feed, the queue will absorb excess concurrency at run-time and thus the response time increases with load but throughput will not degrade. Supporting only one model of concurrency, therefore, will not be efficient or flexible to design highly concurrent real-time systems [6, 49] and severely constrains the programmers [8]. As a consequence, it is conceivable that both programming models should be supported in order to allow the programmer flexibility and efficiency as benefits from both models such as high throughput, ease of facilitating parallelism and programming can be achieved.

2.2 Asynchronous Event Handling Facilities

This section examines event handling facilities provided by high-level programming languages, middlewear, and distributed systems and investigates how they are implemented to help the application's predictable and efficient behaviour, in the context of the concurrent programming models discussed in Section 2.1.

2.2.1 Event Handling in General

An event handler is typically an asynchronous callback that handles raised events. Each event is a piece of application-level information from the underlying framework. The events are created by the framework based on interpreting lower-level inputs, which may be lower-level events themselves. The events initially originate from actions on the operating system level, such as interrupts generated by hardware devices, software interrupt instructions, or state changes in controlling external objects. On this level, interrupt handlers and signal handlers correspond to event handlers. In general, fired (or raised) events are processed by an event dispatcher within the framework. It manages the associations between events and event handlers, and may queue event handlers or events for later processing. Event dispatchers may call event handlers directly, or wait for events to be dequeued with information about the handler to be executed (i.e. priority). Each framework provides a slightly different scheme for event handling. In this subsection some of main stream programming languages are discussed in terms of their event handling approach.

Java

The event handling mechanism used in the original version of Java (1.0) has changed significantly in modern version of Java. In the old Java 1.0 approach, the *inheritance-based event model*, an event was propagated sequentially up the containment hierarchy until either it was handled by a component or the root of the hierarchy is reached. Although the inheritance-based model works fine for small applications, it does not scale very well for large programs. This is mainly because the model wasted CPU time as it required components to receive events in which they are not interested. The modern approach to handling events in Java is based on the *delegation event model* [54], which defined standard and consistent mechanisms to generate and process events. Its concept is quite simple: a source generates an event and sends it to one or more handlers (or listeners in Java). The handler simply waits until it receives an event. Once an event is received, the handler processes the event and then returns. The advantage of the model is firstly that the application logic that processes the event is cleanly separated from the logic of the event source that generates those events, and an event source is able to delegate the processing of an event to a separate piece of code, achieving synchronisation decoupling between participants[2]. Secondly, events are delivered only to components that are interested in the events and therefore general performance (i.e. response-times) is much better (especially in applications with many events or with high-frequency type events).

The delegation event model is comprehensively used for user-interfaces in Java (the Abstract Window Toolkit and the Swing Toolkit). The following explains the general framework to map the delegation event model to the Java API:

- Events: In Java, an event is an object that describes a state change in a source. The classic example of such events is a graphical user interface that generates events

[2]For more information about the decoupling, see Section 2.2.4

as a consequence of a user interacting with the elements of it. Java defines several types of events, such as `ActionEvent` that is generated when a button pressed, `ComponentEvent` that is generated when the size, position, or visibility of a component is changed, `KeyEvent` that is generated when keyboard input occurs, and so on. Events may also occur that are not directly caused by interactions with a user interface. For example, an event may be generated when a timer expires, a software or hardware failure occurs, or an operation is completed. It is also possible to define events that are appropriate for a particular application.

- Event Sources: A source is an object that generates an event. This occurs when the internal state of that object changes in some way. For example the event source, `Button`, generates `ActionEvent` when the button is pressed. Sources may generate more than one type of events and therefore all registered listeners are notified and receive a copy of the event object, *multicasting.* An event source must register handlers (listeners in Java) in order for the listeners to receive notifications about the events generated by the associated event source. The general form of registration method is as the following:

  ```
  public void addTypeListener(TypeListner listener)
  ```

  where *Type* is the name of the event, and *listener* is a reference to the event listener. When such an event occurs, the registered listener is notified. This is known as *unicasting* the event if it has only one registered listener. For a listener to unregister an interest in a specific type of event, the following method is used:

  ```
  public void removeTypeListener(TypeListner listener)
  ```

  For example, to remove a keyboard listener, `removeKeyListener` is called.

- Event Listeners: A listener that is notified when an event occurs is also an object (interface) in Java. It has two prerequisite requirements. First it must be registered with one or more sources to receive notifications about specific types of events. Second, it must implement methods to receive and execute the generated events as listeners in Java are interfaces.

The event handling mechanism used by the Java Swing is also based on the delegation event model. In Java, essentially all event handling is performed by a single thread [67], called the *event dispatch thread.* Therefore handlers are queued in a FIFO manner when associated events are fired and executed by the thread in turn. Therefore it uses the single server event-based programming model via the event dispatch thread. However in Swing, application threads (background threads) can be constructed in the application via two classes, `SwingWorker` and `Timer`, each of which has a associated thread. The `SwingWorker` class can be used to perform time-consuming or blocking events or the `Timer` class to perform periodic events in the background. Listing 2.1 illustrates the code fragment that spawns a separate thread to execute the workload inside the construct method in a `SwingWorker` object. If the classes are used, the mapping between the executionee (a section of code to be executed) and the executioner (server thread) becomes 1:1.

```
SwingWorker worker = new SwingWorker() {
    public void construct() {
        // do something time-consuming or blocking here
    }
};
worker.start();
```

Listing 2.1: A Simple Usage of `SwingWorker`

C# & .Net Framework

Some mainstream programming languages do not contain any built-in support for multi-threaded applications, such as C++[3], or provide asynchronous event handling facilities by its own semantics and specifications, such as C#. Instead they rely entirely upon the operating system (i.e. POSIX, presented in the following subsection) or the library of pre-coded APIs, such as .NET Framework, to provide these absent features. The .NET framework [11] is a software framework that includes a large library and a virtual machine that manages the execution of programs, including the notion of threads and asynchrony. It is also designed for cross-language compatibility and therefore supports many programming languages. They includes Visual Basic, C++, C#, Java, Perl, Python, etc. For example C# [10] is developed mainly to work with .NET Framework and continuously evolves along with it. C# supports event-driven programming via events and delegates. A delegate in C# is similar to a function pointer in C or C++. Using a delegate allows the programmer to encapsulate a reference to a method inside a delegate object. The delegate object can then be passed to code which can call the reference method. Events are declared using delegates in C# and an event is a way for a class to allow clients to give delegates to methods that should be called (i.e. handlers) when the event occurs. When

[3]The new version of the C++ language standard, C++09 [3], is being developed at the time of writing with several new features. The standard is not yet finalised, however it is highly possible that it will support multi-threading library.

events are fired, the delegate(s) given to it by its clients are invoked. In C#, the execution of the method (handler) is performed by the calling thread and is therefore synchronous.

Although the .NET Framework does not directly provide an asynchronous event handling scheme, it recommends the structured way of presenting asynchronous behaviour for classes, called the *Event-Based Asynchronous Pattern*. The pattern provides an effective way to expose asynchronous behaviour with the event and delegate semantics in C#. The pattern is typically referred to as an asynchronous method call. A class that supports the event-based asynchronous pattern has one or more methods named `MethodName`*Async*. These methods may mirror the synchronous counterparts (i.e. without the suffix -*Async*) which perform the same operation on the current thread. The asynchronous version of methods spawns a separate thread that executes the associated handler in the background. The `PictureBox` class in .NET Framework is a typical component that support the pattern. The application can download an image synchronously by calling its `Load` method or asynchronously by calling `Load`'s asynchronous counterpart, `LoadAsync` method, which perform the downloading on a separate thread in the background.

As mentioned earlier events handlers are executed synchronously in C#. However, using the combination of events, delegates and the event-based asynchronous pattern, one could achieve the asynchronous behaviour of event handling. Recall that a reference to a method (handler) is encapsulated inside a delegate object which is called when an associated event occurs. By setting an asynchronous version of method to the associated delegate it is possible to execute the method asynchronously. The relationship between delegates and methods is many to many (therefore has a *multicasting* capability); a delegate object may refer to more than one method and a single method can be associated with more than one delegate object. Delegates in C# are like function pointers and any information required to handle events can be passed to the associated methods (handlers). Like the delegation event model in Java, the modularity and the performance of the application is increased by separating events from the the processing of an event to a separate piece of code providing synchronisation decoupling. However the mapping between the executionee (a section of code to be executed) and the executioner (server thread) is 1:1 using this approach.

2.2.2 Real-Time Signals in POSIX Real-Time Extensions

POSIX (or Portable Operating System Interface) [31] is the collective name of related standards specified by the IEEE to define the application programming interface (API), along with shell and utilities interfaces for software compatible with variants of the Unix operating system, although the standard can apply to any operating system. POSIX 1003.1 defines the basic functions of a Unix operating system, including process management, device and file system I/O, and basic IPC (Inter-Process Communication) and time services. Most of modern general-purpose operating systems support this standard. For example most Linux distributions support it via the Linux Standard Base (LSB) [20]. The real-time (1003.1b) and thread (1003.1c) extensions of POSIX 1003.1 define a subset of POSIX interface functions that are particularly suitable for multi-threaded real-time applications. Specifically, POSIX 1003.1c extends 1003.1 to include the creation of threads and management of their execution within a process. Real-time extensions defined by 1003.1b include prioritised scheduling, enhanced real-time signals, IPC primitives, high-resolution timer, memory locking, synchronous I/O, asynchronous I/O, and contiguous files. Although there are other extensions to POSIX 1003.1 such as 1003.1d (additional real-time extensions) and 1003.1j (advanced real-time extensions), the first two standards (1003.1b and 1003.1c) are most widely supported in many real-time operating systems.

POSIX Signals

Signals in POSIX are primarily used as a notification mechanism when an important event has occurred in the system. Examples of such events are exception handling or asynchronous interrupts. Signal delivery in POSIX is on per process basis. A process may have one or more concurrently active threads inside. A process can signal another process to synchronise and communicate. A process has a service function, called a signal handler. When the system delivers a signal to the process, the signal handler is executed. Therefore the signal mechanism provides asynchrony and immediacy, just as hardware interrupts do.

There is a number of pre-defined signals, each of which is allocated an integer value. Most POSIX signals are used by the operating system; what they do is implementation-defined by the system. There are also a number of implementation-defined signals for application use. Each signal has a default handler, which usually terminates the receiving process. Example signals are: `SIGBART` for abnormal termination, `SIGALRM` for alarm clock expiry, `SIGKILL` for illegal instruction exception, `SIGTERM` for terminating a process,

```
union sigval{
   int sival_int;
   void *sival_ptr;
};

struct sigevent{
   int sigev_notify;
   int sigev_signo;
}

typedef struct{
   int si_signo;
   int si_code;
   union sigval si_val;
}siginfo_t;

typedef ... sigset_t;

struct sigaction {
   void (*sa_handler)(int signum);
   void (*sa_sigaction)(int signum,
                        siginfo_t *data,
                        void *extra);
   sigset_t sa_mask;
   int sa_flags;
}

int sigaction(int     signum,
              const  struct sigaction *reaction,
              struct sigaction *old_reaction);

int sigsuspend  (const sigset_t *sigmask);
int sigwaitinfo (const sigset_t *set, siginfor_t *info);
int sigtimedwait(const sigset_t *set, siginfor_t *info,
                 const struct timespec *timeout);

int sigprocmask(int how, const sigset_t *tset, sigset_t *oset);
// manipulates a signal mask according to the vaule of how
// how = SIG_BLOCK   -> the set is added to the current set
// how = SIG_UNBLOCK -> the set is subtracted from the current set
// how = SIG_SETMASK -> the given set becomes the mask

int sigemptyset(sigset_t *s);
int sigfillset(sigset_t *s);
int sigaddset(sigset_t *s, int signum);
int sigdelset(sigset_t *s, int signum);
int sigismember(const sigset_t *s, int signum);

int kill(pid_t pid, int sig);
int sigqueue(pid_t pid, int sig, const union sigval val);
```

Listing 2.2: An Interface to POSIX Signals in C

SIGFPE for floating-point exception, and SIGPIPE for writing a pipe without readers (signals can also be identified by numbers and the correspondence between the number and the symbolic name of each signal, that is given in the header file <signal.h>). Listing 2.2 [6] shows the specification of a C interface to the POSIX signal interface. There are also signals which are to be used solely for application-defined purposes. An application-defined signal (indicated by one of the numbers reserved for applications) is defined by the function (i.e. the signal handler) which application sets up for the system to call back when a signal with that number arrives. The action to be performed upon the occurrence of a signal is specified by the data structure sigaction on the signal (i.e. sa_handler for a non real-time handler and sa_sigaction for a real-time handler). The information provided by the data structure also includes a mask sa_mask and a flag field sa_flag. A signal handler can be set up using the function sigaction with other arguments of it.

Standard POSIX signals can be generated via the kill function. A process can also request that a signal be sent to itself: when a timer expires (i.e. SIGALRM), when asynchronous I/O completes, by the arrival of a message on an empty message queue, or by using the C raise statement. POSIX maintains the set of signals that have been currently masked by a process. The function sigprocmask is used to manipulate this set. The how parameter is set to: SIG_BLOCK to add signals to the set, SIG_UNBLOCK to subtract signals from the set, or SIG_MASK to replace the set. The other two parameters contain pointers to the set of signals to be added/subtracted/replaced (set) and the returned value of the old set (oset). Various functions (sigemptyset, sigfillset, sigaddset, sigdelset, and sigismember) allow a set of signals to be manipulated. There are three ways in which a process can deal with a signal based on the mask applied to the set:

- It can handle the signal by setting a function to be called whenever it occurs, which is the default behaviour. The signum parameter in the sigaction function in Listing 2.2 indicates which signal to be handled, reaction is a pointer to a structure containing information about the handler, and old_reaction points to information about the previous handler[4]. The sa_handler member in the sigaction data structure indicates the action associated with the signal and can be SIG_DFL (default action, usually to terminate the process), SIG_IGN (ignore the signal), or a pointer

[4]Essentially, the information about a handler contains a pointer to the handler function (sa_handler if the signal is a non-real-time signal, or sa_sigaction if the signal is a real-time one), the set of signals to be masked during the handler's execution (sa_mask), and whether the signal is to be queued (indicated by setting sa_flags to the symbolic constant SA_SIGINFO).

to a function (the code that gets executed when the signal is delivered).

- It can ignore the signal altogether (in which case the signal is simply lost) except those used by the system to stop and destroy an errant process (e.g. `SIGKILL`). This can be achieved by setting the value of `sa_handler` to `SIG_IGN` in a call to the function `sigaction`.

- It can block the signal and either handle it later or accept it. Just as it is necessary to be able to disable interrupts, it is essential that signals can be selectively blocked from delivery. When a signal is blocked by adding the value of the signal to block to the set, it remains pending until it is unblocked (i.e. removing its number from the signal mask) or accepted. A signal is said to be accepted by a process (or thread) when the signal is selected and returned by one of the `sigwait` functions. When it is unblocked, it is delivered.

Essentially signal delivery is on per process basis, hence if a process ignores a signal, all threads in the process do as well. However each thread has its own signal mask. Therefore, threads in a process can choose to block or unblock signals independent of other threads. Some signals are intended for the entire process. Such signals, if not ignored, are delivered to any thread that is in the process and does not block the signal. A signal is blocked only when all threads in the process block it.

POSIX Real-Time Signals

POSIX real-time extensions [31] extend the standard POSIX signals and the traditional Unix signals to make the mechanism more responsive and reliable. The following lists the improvements of the POSIX real-time signal [6, 45]:

- There are at least 8 application-defined real-time signals versus only two provided by the standard POSIX. These signals are numbered from `SIGRTMIN`, the identifier of the first signal, to `SIGRTMAX`, the identifier of the last signal. Only those signals whose numbers lie between the two are considered to be real-time by POSIX.

- Real-time signals, whose value lie between `SIGRTMIN` and `SIGRTMAX`, can be queued, while traditional Unix signals cannot. Queuing is a per signal option; one chooses the queuing option for a real-time signal by setting `SA_SIGINFO` in the `sa_flags` field of the `sigaction` structure of the signal. If not queued, a signal that is delivered while being blocked may be lost. Hence, queuing ensures the reliable delivery of signals.

- Real-time signals can carry more data via a pointer to the `siginfor_t` structure while a traditional signal has only one parameter: the number of the signal. The additional structure contains the signal number, a code which indicates the cause of the signal (for example, a timer signal) and an integer or pointer value. This capability increases the communication bandwidth of the signal mechanism. As an example, a server can use this mechanism to notify a client of the completion of a requested service and pass the result back to the client at the same time.

- Real-time signals can be delivered in a priority-ordered manner: The smaller the signal number, the higher the priority (i.e. `SIGRTMIN` is delivered before `SIGRTMIN` + 1 and so on).

- Real-time extensions provide a new and more responsive synchronous signal-wait function called `sigwaitinfo`. The function suspends the calling process until the signal arrives. To use this function, the signal must be blocked, and thus the handler is not called when it is delivered. Instead, the function returns the selected signal number and stores the information about the delivered signal in the `info` argument. The function `sigtimedwait` has the same semantics as `sigwaitinfo`, but allows a timeout to be specified for the suspension. If no signals are delivered by the timeout, `sigtimedwait` returns -1 and `errno` set to `EAGAIN`. In contrast, the POSIX synchronous signal-wait function `sigsuspend` remains blocked until the signal handler executes.

Implementation of POSIX Real-Time Signals

The original POSIX signal model came from Unix and was extended to make it more appropriate for real-time when the real-time extensions to POSIX were specified. With the POSIX Thread Extensions, where the execution entities are threads instead of processes, the model has become more complex and is a compromise between a per process model and a per thread model. The complexity stems from selecting the target thread of a signal. For single-threaded processes, the model determining the target process of a signal is quite simple and straightforward, but for multi-threaded processes the implementation of the model is undefined by POSIX. For example if more than one thread is eligible to have a signal delivered to them, it is not defined which thread is chosen. Of course this can be avoided if a programmer makes sure that only one thread is eligible for a specific signal. However this will make the application much less flexible. In [15], the notion of *signal owner* is proposed to eliminate the indeterminism associated multi-threaded processes when implementing the POSIX real-time signal. The signal owner is defined as the thread that installed the signal handler last, or that made a call to any of `sigwait` related functions last. Therefore determining the receiving thread with the signal owner model is straightforward and reduces the overhead of checking every thread's signal mask to find the suitable thread to deliver the signal. However the following points should be noted [6] when implementing or using the POSIX real-time signal:

- Signals generated as a result of a synchronous error condition, such as memory violation and erroneous arithmetic operation, are delivered only to the thread that caused the signal.
- Other signals may be sent to the process as a whole; however, each one is only delivered to a single thread in the process.
- The `sigaction` function sets the handler for all threads in the process.
- The function `kill` and `sigqueue` can be applied to both processes and threads. A new function `pthread_kill` allows a process to send a signal to an individual thread.
- If the action specified by a handler for the signal is for termination, the whole process is terminated, not just the receiving thread.
- Signals can be blocked on a per thread basis using a function `pthread_sigmask` which has the same set of parameters as `sigprocmask`. The use of the function `sigprocmask` is not specified for a multi-threaded process.
- The functions `sigsuspend`, `sigwaitinfo` or `sigtimedwait` operate on the calling thread not the calling process.
- A new function `sigwait` allows a thread to wait for one of several blocked signals to occur. It behaves the same as `sigwaitinfo` except that the information associated with the signal is not returned. The signals are specified in the location referenced as a parameter. The function returns zero when a successful wait has been performed and the location referenced by `sig` contains the received signal. If more than one real-time signal is pending, the one with the lowest value is chosen.
- If a signal action is set by a thread to 'ignore', it is unspecified whether the signal is discarded immediately after it is generated or remains pending.

Although there are some ambiguity in the implementation of POSIX real-time signals as the above, the mapping between signal handlers and their execution entities (i.e.

threads) is 1:1. In other words, a signal handler will always be executed by one of threads that has installed the handler with the relevant signal of the handler in which the thread is interested. If there is no threads to handle a delivered signal (i.e. all threads block delivery of the signal), the signal remains pending until a thread unblocks the signal. This greatly reduces the complexity of implementation due to the following two criteria:

- Dynamic mapping between signal handlers and threads are not required (as it is, to some extent, static - threads register signals in which they are interested and signals are delivered to the threads that registered the signals only),
- Priority changes of server threads can be bypassed if handlers are assigned to server threads at the same priority level, and
- Run-time threads creation to execute signal handlers is avoided (even if there is no threads to handle a signal).

2.2.3 Ada 2005

Ada [59] is object-oriented, real-time programming language that is intended for used in large, long-lived applications where reliability and efficiency are essential, particularly real-time and embedded systems. It is also the only ISO standard programming language and has been continuously evolved from its inception (Ada 83, Ada 95, and the current version Ada 2005). Unlike the first version of the language, Ada 83, which defined a single language, the Ada 95 and 2005 definitions have a core language design plus a number of domain-specific annexes, which include systems programming and real-time programming. Ada 2005 adds further improvements to Annex D: Real-Time Systems Annex [59] by the addition of advanced paradigms for scheduling and timing. Particularly it enables the development of real-time programs with predictable behaviour. The core Ada language says little about scheduling and priorities and hence, for example, entry queues[5] have to be serviced in order of arrival. However the real-time systems annex defines various scheduling policies in terms of priorities (i.e. EDF and Round Robin within priorities), which gives more predictable behaviour and enables schedulability analysis. Note that to conform to the real-time systems annex, an implementation must also conform to the systems programming annex which covers access to machine code, interrupt handling, some extra requirements on representations and preelaboration, a pragma for discarding names

[5]The keyword `Entry` in the Ada language is used as a rendezvous mechanism among tasks and an entry queue denotes the data structure that lists waiting tasks (or clients). If the Ada implementation does not support the real-time annex, the order of the queue is FIFO.

at run time, the pragmas for shared variables and packages for general task identification, per task attributes and the detection of task termination.

Events and event handling are newly introduced features in Ada 2005 for the real-time systems annex and the event handling facilities work essentially with timers (i.e. an event is triggered by the passage of time, not by an action and such external and internal actions must be handled by other means such as interrupts or asynchronous transfer of control). There are four types of implicit and explicit events: Two are concerned with monitoring the CPU time of tasks (The `Timers` package applies to a single task and the `Group_Budgets` package to groups of tasks). The `Task_Termination` package is linked to a termination of a task. The `Timing_Event` package represents timers that measure real time and are used to trigger events at specific real times. Although there are some differences between these events due to their particular application context, they all essentially have the same behaviour.

```
package Ada.Real_Time.Timing_Events is
   type Timing_Event is tagged limited private;
   type Timing_Event_Handler is access protected procedure
        (Event : in out Timing_Event);
   procedure Set_Handler (Event    : in out Timing_Event;
                          At_Time  : in Time;
                          Handler  : in Timing_Event_Handler);
   procedure Set_Handler (Event    : in out Timing_Event;
                          In_Time  : in Time_Span;
                          Handler  : in Timing_Event_Handler);
   function Current_Handler (Event : Timing_Event)
        return Timing_Event_Handler;
   procedure Cancel_Handler (Event      : in out Timing_Event;
                             Cancelled  : out Boolean);
   function Time_Of_Event (Event : Timing_Event) return Time;
private
   ... -- not specified by the language
end Ada.Real_Time.Timing_Events;
```

Listing 2.3: The `Ada.Real_Time.Timing_Events` Package Specification

Timing Events

Timing events are supported by a child package of `Ada.Real_Time` as illustrated in Listing 2.3. `Timing_Event` represents a time in the future when an event to occur and is a `tagged limited private` type. The event type being tagged indicates that values of the type carry a tag at run time (polymorphism via *dynamic dispatching operations* or *late binding* - deciding which procedure to dispatch at run time based on the parameter passed in) and that the type can be extended (inheritance). The keyword `limited` type means that only those subprograms defined in the same package can be applied to the type. The type `Timing_Event_Handler` identifies a protected procedure to be executed by the implementation when the timing event occurs and therefore is an access to the protected procedure with the timing event itself being passed back to the handler when the event is triggered. The event is said to be *set* if it is associated with a handler and *cleared* otherwise. Two `Set_Handler` procedures are defined, one using absolute time, the other relative; both use the clock within package `Ada.Real_Time` as the reference clock. A call of `Set_Handler` for an event that is already set replaces the handler and the time of execution. The function `Current_Handler` returns the handler associated with the event if it is set or null otherwise. `Cancel_Handler` clears the event if it is set and this can also be achieved by calling `Set_Handler` with passing a null event. Hence the relationship between handler and event is 1:1. The function `Time_of_Event` returns the time of the event. As soon as possible after the time set for the event, the handler is executed, passing the event as parameter. The handler is executed only if the timing event is in the set state at the time of execution and this clears the event (i.e. it will not be triggered again unless it is reset).

Implementation of Timing Events

The Ada 2005 Reference Manual (ARM) [59] recommends that the handler procedure should be executed directly by the real-time clock interrupt mechanism (i.e. interrupt handler of the reference clock). This is the most effective way for an implementation to support timing events as the clock interrupt handler typically runs at the highest priority in the system [7] and is executed directly by the system. Therefore the protected object that embodies the handler procedure must have a ceiling of the highest interrupt priority (i.e. `Interrupt_Priority'Last` which denotes the last interrupt priority value in the scalar type `Interrupt_Priority`). The framework for event handling in Ada 2005 provides a lightweight facility for reacting to the arrival of a point in time and therefore avoids the concurrency-related overhead. Again as for the POSIX real-time signals, this single server thread approach greatly reduces the complexity of implementation due to the following two criteria:

- Dynamic mapping between handlers and threads are not required (as they are typically executed by the system with the highest interrupt priority and hence no priority change is required),

- Run-time threads creation to execute handlers is avoided.

2.2.4 Event Handling in Distributed Systems

As the cost of microprocessors and communication technology has continually decreased in real terms, distributed computer systems become a viable alternative to uniprocessor and centralised systems in many embedded application area for reasons of reliability and performance [6, 40]. Distributed system can be defined to be a system of multiple autonomous processing elements cooperating in a common purpose or to achieve a common goal. These autonomous processing elements are in general physically dispersed. In such systems, events play a major role for exchanging information to facilitate communication among these physically dispersed elements in the system. To effectively control the environment, their communication must be efficient and reliable.

Recent trends towards server/client applications and open distributed systems are resulting in an increasing number of *middleware*[6] products as object-oriented technology has become very popular. The concept of a middleware was first introduced to facilitate communication between entities in a heterogeneous distributed computing environment. As one of the main responsibilities of a middleware is to manage the communication of distributed components in the system; it can often be classified according to its communication model as follows [19]:

- Synchronous communication model: message passing, remote procedure calls (RPC), notifications (or observer design pattern) (i.e. used in CORBA)
- Asynchronous communication model: tuple spaces, message queuing, publish/subscribe (i.e. used in Real-Time Publish/Subscribe model)

The synchrony/asynchrony used to distinguish the communication models above is based on *synchronisation decoupling.* That is producers are not blocked while producing events, and consumers can be asynchronously notified of the occurrence of an event while performing some concurrent activity. The production and consumption of messages do not happen in the main flow of control of the producers and consumers, and do not therefore happen in a synchronous manner. *Space* and *time decoupling* should also be taken into

[6]Large scale open distributed systems are generally build on top of a set of distribution mechanisms packaged into a layer known as middleware and this technique, isolating the application from platform specific differences both hardware and software and providing facilities to hide the undesirable aspects of distribution, is often called *distribution transparency.*

account in order for the distributed system to be more efficient and hence scalable. Space decoupling indicates that the interacting parties do not know each other. The producers publish events through an event service and the consumers get the events indirectly through the event service. The producers and consumers, therefore, do not hold references to each other. Time decoupling shows that the interaction between producers and consumers do not need to be actively participating in the interaction at the same time. In particular the producer may publish events while the consumers are disconnected, and conversely, the consumer might get notified about the occurrence of the events, for example, while the producer of the events is disconnected. Decoupling of the above three criteria removes all explicit dependencies between the interacting participants. This decoupling enable the resulting communication application to be well adapted to distributed environments that are asynchronous. This should be noted that event handling approaches presented in this section represents a higher level of abstraction for communication between distributed entities. This differs from event handling approaches in centralised systems discussed in the previous section, although there are some similarities (e.g. their classification is based on the synchronisation decoupling). However delivered messages or events must be handled using a certain event handling approach in the receiver's perspective. In this section, therefore, communication models used in some well-known distributed systems are presented first and then an exemplary event handling implementation for distributed systems with respect to its dispatching scheme is discussed in the last sub-section.

Common Object Request Broker Architecture (CORBA) Event Service

As a CORBA extension, the CORBA Event Service [25] is introduced to alleviate some of restrictions placed on the standard CORBA's synchronous request/replay communication model. The communication model used in the standard CORBA is remote procedure (or method) calls and therefore by its nature introduces a strong time, synchronisation, and space coupling. The CORBA Event Service defines sets of interfaces and modules to support asynchronous event delivery and many-to-many message delivery. The new CORBA architecture is centred around the concept of an *event channel*, which allows multiple suppliers to communicate with multiple consumers asynchronously in a decoupled way. Event channels are implemented using standard CORBA objects and communication with an event channel for consumers and suppliers is accomplished using either the *push* and *pull* models of the communication. As their names imply, the push model allows an active supplier to push an event to a passive push consumer and in the pull model, a passive supplier waits for an active pull consumer to pull an event from it. As suppliers and consumers are decoupled by an event channel, the push and pull models can be mixed in a single system. The CORBA Notification Service Specification [26] extends the Event Service to provide new capabilities for flexible and accurate event delivery (event filtering and QoS). The asynchronous communication of the Event Service is implemented on top of CORBA's standard synchronous method invocation and thus the substantial overhead of performing a remote method invocation is incurred for every event communication. The Notification Service suffers from the same problems as the Event Service in order to maintain backwards compatibility. The architecture of an event channel implementation is discussed in more detail later in this chapter.

CORBA Real-Time Specification

The standard CORBA lacks in the desirable features for developing real-time applications, such as priority-based real-time event dispatching and scheduling, periodic event processing, centralised event filtering, and correlations. The CORBA Real-Time Specification [24], or RT-CORBA, extends the standard CORBA to support the quality of service requirements and end-to-end predictability. The RT-CORBA 1.0 standard is designed for fixed priority scheduling and RT-CORBA 2.0 targets dynamic scheduling. It provides *distributable threads* as a first-class abstraction to achieve the real-time requirements. A distributable thread can extend and retract its locus of execution points among operations in object instances across physical components by location independent invocations and (optionally) returns. Within each node, the flow of control is equivalent to normal local thread execution. Distributable thread-based programming models imply the need for a number of supporting facilities; thread control actions (e.g. suspend, resume, abort, time constraint change, etc.) and mechanisms that may have to be provided to support maintaining correctness of distributed execution, and consistency of distributed data. All of these facilities generally would be required in order for the system to be timely (e.g. subject to completion time constraints). These supporting facilities, however, are not addressed in the RT-CORBA specification. It still uses the CORBA Event Service as its primary communication model that is synchronous.

Distributed Real-Time Specification for Java

The development of the RTSJ has spurred on the interest for a distributed real-time Java platform, resulting in extending the RTSJ [5] in a familiar way for Java programmers to distributed real-time systems by enhancing the Java RMI to support application-specific predictability of end-to-end timeliness for distributed applications. This work is under progress as Java Specification Request 50 (JSR-50) [23] and is called *Distributed Real-Time Specification for Java*, or DRTSJ. As Java is a well-constructed multi-threaded language, the DRTSJ also adapts the notion of distributable threads as the RT-CORBA does and use them as a basic building blocks for execution and communication in the distributed environment. It however treats asynchronous event handling as a concomitant facility and hence, it has to be provided by some other means, such as vendor-specific added value class libraries. The RTSJ's asynchronous event handling, that is presented in Section 2.3, will be maintained solely for the local, not distributed, implementations of scheduling framework primitives as the DRTSJ Expert Group have not yet made any commitment about their appearance of distributed extensions in the DRTSJ [1].

Real-Time Publish/Subscribe Model

Although RT-CORBA enhanced capability to support asynchronous and many-to-many communication model, it is realised by building on top of their existing communication service that only support synchronous request/reply communication model. This synchronous execution of an operation results in higher overhead imposed on the enhanced communication service. The Real-Time Publish/Subscribe model proposed in [51] follows the pure asynchronous many-to-many communication paradigm. This model attempts to address some of the issues associated with synchronous request/replay communication model in addition to the need to maintain analysability, scalability and efficiency for real-time purposes. It associates *logical handles*, also called *distribution tag* describing the subscription of a message, to each message type. Messages can be generated and received based only on the logical handles regardless of their source(s) or destination(s). Consumers can subscribe to and receive all messages published with that logical handle. Therefore, publishers and subscribers need not know each other. However, it also provides information about source/destination of the message on a certain message type *if needed*. Each distributed node has its own daemon consisting of three threads; Local manager, Update manager, and Delivery manager. These managers are prioritised, giving the highest priority to the manager that deals with routing the messages. However, the messages themselves are not prioritised and hence they are arrived and processed in a FIFO manner. The Object Management Group recently released the *Data Distribution Service Specification for Real-Time Systems (or DDS)* [27], based on the real-time Publish/Subscribe communication model. The specification is developed with the needs of real-time systems in mind and uses the Publish/Subscribe model comprehensively to achieve asynchrony (i.e. data filtering is done in both topic- and content-based and data exchanging is managed

Abstraction	Space decoupling	Time decoupling	Synchronisation decoupling
Message Passing	No	No	Producer-side
Remote Procedure Calls	No	No	Producer-side
Notifications	No	No	Yes
Tuple Spaces	Yes	Yes	Producer-side
Message Queuing	Yes	Yes	Producer-side
Publish/Subscribe	Yes	Yes	Yes

Table 2.1: Decoupleness of Communication Models

by *Global Data Space* which act as a event channel in the RT-CORBA). It also defines a extensive list of QoS parameters, via the `QoSPolicy` class, that allows the application to finely tune the resources used by the system and ensures predictable behaviour.

Implementation of Centralised vs Distributed Systems

As discussed earlier, event handling in distributed systems is quite different from that in uniprocessor and centralised systems. It requires additional functionalities to deal with the *loosely coupled* and *heterogeneous* nature of distributed systems. The communication paradigm provides an abstraction that describes how remote applications (or objects) can communicate with one another in the distributed environment. The communication models are generally classified based on the level of asynchrony (or decoupleness) they provide. They include message passing, remote procedure calls, notifications, shared spaces, message queuing, and publish/subscribe. In [19] these communication models are compared based on the level of decoupling, presented in Table 2.1. By nature, synchronous communication models introduce a strong time, synchronisation (on the consumer side), and also space coupling. As a result, they produce higher latency of remote calls and larger failure rate of event delivery than that of small distributed systems. Only the publish/subscribe model provides full decoupling of both sides, producers and consumers, to enable the distributed system to be more scalable and flexible.

As a part of the TAO research project [56] conducted in Washington University, an implementation of the Real-Time CORBA 1.0 and 2.0 along with a Real-Time CORBA Event Service [28, 41, 57] has been developed and used in a wide range of distributed real-time systems. The scheduling schemes in TAO are performed with a standard *thread pool model*. Threads (i.e. distributable) are schedulable entities in the RT-CORBA and

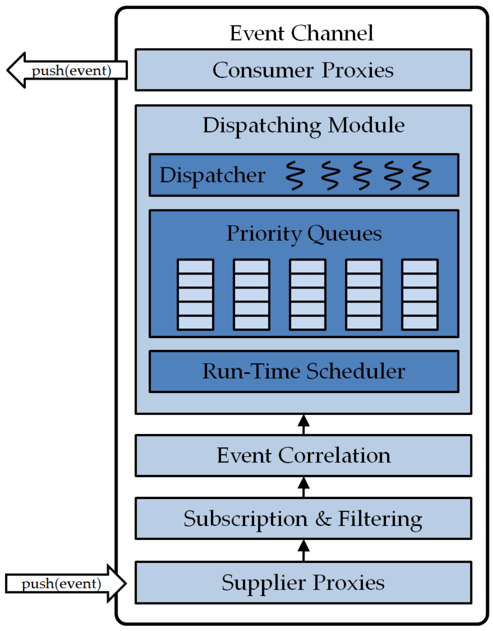


Figure 2.2: RT-CORBA Event Channel Dispatching

can be prioritised. Server applications can specify the default number of static threads, maximum number of threads, and the default priority of all these threads. If an event arrives and all existing threads are busy, a new thread may be created to handle the event unless the maximum number of threads in the pool have been reached. Furthermore, buffering can be enabled using priority queues at transient overload situations in order for the event to be queued until a thread is available to process it. As shown in Figure 2.2, the dispatching module of pending events is decoupled with the run-time scheduler for allowing scheduling policies to evolve independently of the dispatching mechanism which is fully priority-based preemptive. The run-time scheduler determines two priority values of the events, a preemption priority and a sub-priority, according to the specific scheduling policy defined by the event channel. The preemption priority is used to insert the event onto the queue and the sub-priority is used by the dispatcher to determine where to put the event in the priority queue. The *dispatcher* in Figure 2.2 is responsible for forwarding the events to consumers by calling the `push` operation (i.e. depending on the priority of the consumer for which the events are destined). This strategy ensures the consumers execute in time to meet their deadlines. TAO's Event Channel Dispatching Module also supports other levels of concurrency models when dispatching events (i.e. single-threaded priority-based dispatching).

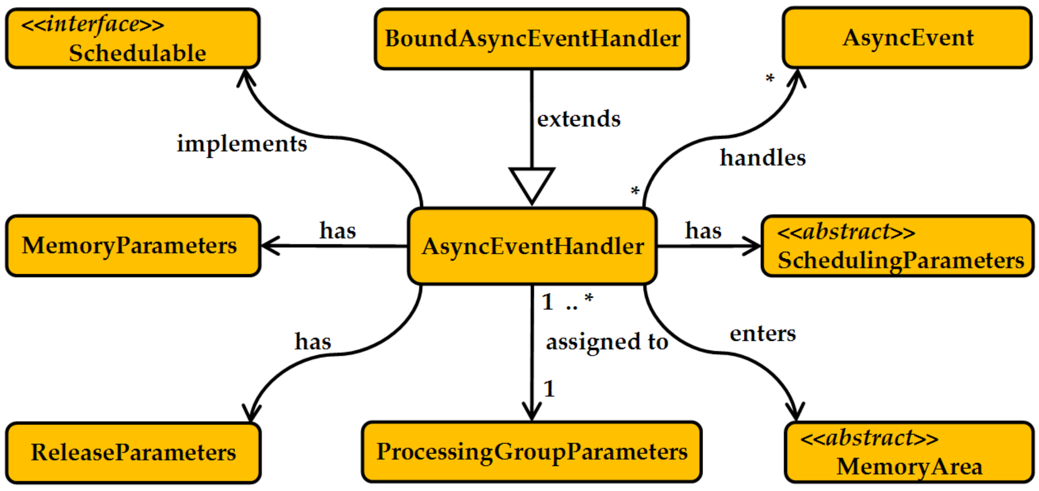


Figure 2.3: AEH Facilities in the RTSJ

Although the RT-CORBA provides comprehensive specifications to implement scalable and flexible middleware, it leaves the mapping between objects (i.e. messages and events) and execution entities (i.e. threads) as a quality of implementation issue. For example TAO adapts a standard thread pool model as its events dispatching module, using Leader/Followers [55] concurrency design pattern, which essentially assigns a single thread (a dispatcher) to an arrived event (i.e. a dynamic 1:1 mapping)[7]. Other implementations of distributed communication models take the similar approach as the TAO's ORB Core module. When distributable threads are used, the implementation of the mapping becomes simpler as a reply/request communication is executed by a single distributable thread across distributed nodes. However the distributable thread also has to be mapped to and executed by a local thread. The mapping between the distributable and the local threads becomes an issue again. However it is widely accepted and secure practice to map the two entities 1:1 to alleviate concurrency related problems [28].

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

Figure 2.3 illustrates the classes and interface in the RTSJ related to asynchronous event handling. There are two types of schedulable objects that implement the `Schedulable` interface: the `AsyncEventHandler` and the `RealtimeThread` class. Schedulable objects in the RTSJ is the generalised form of entities that can be scheduled. Each schedulable object may have a set of parameters that control their actual execution. The set of parameters specifies timing, scheduling, memory, and thread group requirements for its associated instance via `Release`, `Scheduling`, `Memory`, and `ProcessingGroupParameters`, respectively.

- The `ReleaseParameters` class is a base abstract class of periodic, sporadic and aperiodic release parameters. It characterises how often a schedulable object is released and gives the general timing parameters that all schedulable objects require. The timing parameters include an estimate of the worst-case execution time (cost) and deadline, along with handlers for their cost overrun (optional in the RTSJ) and deadline miss. A null handler can also be set to indicate that it is not concerned with a missed deadline or a cost overrun.
- The `SchedulingParameters` class is a null abstract class in order to be fully extensible [67] and there are two subclasses of it defined by the RTSJ, priority parameters and importance parameters. The RTSJ requires its implementations to support at least 28 unique real-time priority levels. The importance parameters are not used by the RTSJ default priority-based scheduler which uses the priority parameters only. However the importance parameters can be used, for example, to inform the scheduler which schedulables are more critical when the system enters into a transient overload situation. Note that the importance parameters is used to denote the notion of preemption threshold of a handler to apply fixed priority preemptive scheduling with preemption threshold to AEH, which is presented in Section 5.3.
- The `MemoryParameters` class provides the maximum amount of memory used by the object in its default memory area and in immortal memory[8], and the maximum

[7]For more information about the Leader/Followers design pattern, see Section 3.1.4.

[8]The RTSJ provides two alternative to using the heap memory area. One is immortal memory which can never be automatically reclaimed and hence can avoid the interference from the garbage collector. The other is scoped memory which have a well-defined lifetime. Schedulable objects may enter and leave a

allocation rate of heap memory. When a schedulable object which has memory parameters exceeds its allocation or allocation rate limit, the error is handled as if the allocation failed because of insufficient memory. The object allocation throws an `OutOfMemoryError`.

- The `ProcessingGroupParameters` class allows a set of schedulables to be treated as a group and to have an associated period, cost and deadline. Therefore it is mainly used to bound aperiodic schedulables' impact on the overall schedulability of the system by implementing an aperiodic server technique [71].

`RealtimeThreads` and `AsyncEventHandlers` are both `Schedulable` objects in the RTSJ in order to provide a framework from within which on-line feasibility analysis of priority based systems can be performed for uniprocessor systems. However their detailed semantics, usage, and design goals are different. In this section, the RTSJ event handling facilities is discussed in detail. The basic event model is presented first, followed by a priority-based implementation of asynchronous event handling.

2.3.1 The Basic Model

```
public class AsyncEvent {
   /* constructor */
   public AsyncEvent();
   /* methods     */
   public void addHandler(AsyncEventHandler handler);
   public void removeHandler(AsyncEventHandler handler);
   public void setHandler(AsyncEventHandler handler);
   public void fire();
   public boolean handledBy(AsyncEventHandler handler);
   public void bindTo(String happening);
   public void unbindTo(String happening);
   public ReleaseParameters createReleaseParameters();
}
```

Listing 2.4: The `AsyncEvent` Class

The RTSJ provides three essential classes to facilitate asynchronous event handling. They are `AsyncEvent` (AE or event), `AsyncEventHandler` (ASEH or handler), and `BoundAsync-EventHandler` (BASEH or bound handler). The `AsyncEvent` class is given in Listing 2.4. The `addHandler` and `removeHandler` methods add and remove a handler to the set of

scoped memory area and when there are no schedulables active inside a scoped memory area, the allocated memory is automatically reclaimed. When a schedulabe object is created, its initial memory area can be set via `MemoryArea` in Figure 2.3.

handlers associated with the event, respectively. Therefore an event can have more than one associated handler. On the other hand, the `setHandler` method associates a new handler with the event and remove all existing handlers. The `fire` method fires the instance of `AsyncEvent`. When the method is called, the `fireCounts` of the associated handlers are incremented and all attached handlers are released. The execution of first three methods, `addHandler`, `removeHandler`, and `setHandler` is atomic with respect to the execution of the `fire` method. That is handlers cannot be added, removed, or set while the event is releasing the current set of associated handlers. The `bindTo/unbindTo` method binds/unbinds the instance of `AsyncEvent` to an external event, a *happening*. The values of happening are implementation-defined. An instance of `AsyncEvent` is considered to have occurred (or fired) whenever the happening is triggered. The `createReleaseParameters` method returns the release parameters associated with this event. The information can then be used to fill in the parts of the release parameters of associated handlers (e.g. cost).

```
public class AsyncEventHandler implements Schedulable {
   /* Constructors */
   public AsyncEventHandler();
   public AsyncEventHandler(boolean nonheap);
   public AsyncEventHandler(boolean nonheap, Runnable logic);
   public AsyncEventHandler(Runnable logic);
   public AsyncEventHandler(SchedulingParameters scheduling,
                            ReleaseParameters release,
                            MemoryParameters memory,
                            MemoryArea area,
                            ProcessingGroupParameters group,
                            boolean nonheap);

   .. other constructors are also available

   /* AEH Methods                        */
   /* 1. Schedulable interface methods */
   public boolean addIfFeasibility();
   public boolean addToFeasibility();
   public boolean removeFromFeasibility();
   public boolean setIfFeasible(ReleaseParameters release,
                                MemoryParameters memory);

   .. other setIfFeasible methods are available

   public boolean setSchedulingParametersIfFeasible(
                                SchedulingParameters scheduling);
   public boolean setReleaseParametersIfFeasible(
                                ReleaseParameters release);
   public boolean setMemoryParametersIfFeasible(
                                MemoryParameters memory);
   public boolean setProcessingGroupParametersIfFeasible(
                                ProcessingGroupParameters group);
   public MemoryArea getMemoryArea();
   public MemoryParameters getMemoryParameters();
   public ProcessingGroupParameters getProcessingGroupParameters();
   public ReleaseParameters getReleaseParameters();
   public Scheduler getScheduler();
   public SchedulingParameters getSchedulingParameters();
   public void setMemoryParameters(MemoryParameters memory);
   public void setProcessingGroupParameters(
                                ProcessingGroupParameters group);
   public void setReleaseParameters(ReleaseParameters release);
   public void setScheduler(Scheduler scheduler);

   .. other setScheduler methods are available

   /* 2. Daemon status mutator & accessor methods */
   public final boolean isDaemon();
   public final void setDaemon(boolean on);

   /* 3. Event handling methods */
   protected int getPendingFireCount();
   protected int getAndClearPendingFireCount();
   protected int getAndDecrementPendingFireCount();
   protected int getAndIncrementPendingFireCount();
   public void handleAsyncEvent();
   public final void run();
}
```

Listing 2.5: The `AsyncEventHandler` Class

An `AsyncEventHandler` is a `schedulable` object encapsulating code that is released for execution in response to the occurrence of an associated event. Each handler behaves as if it is executed by a `RealtimeThread` or `NoHeapRealtimeThread`. There is not necessarily a separate real-time thread for each handler, but the server real-time thread (returned by `currentRealtimeThread`) remains constant during each execution of the `run` method. The `AsyncEventHandler` class is alleviated in Listing 2.5. It has several constructors that allow the passing of various parameter classes associated with a schedulable object (scheduling, release, memory, processing group, and memory area parameters as discussed earlier) along with the optional `Runnable` object. By setting the `nonheap` boolean parameter to true the handler may not access the default heap memory. Application-defined handlers can be created either by subclassing the base class and overriding the `handleAsyncEvent` method or by passing a runnable object to one of the constructors. The methods in the `AsyncEventHandler` class can be divided into three categories as follows (See Listing 2.5):

1. `Schedulable` interface methods which needed to support the schedulable interface,

including the accessor and mutator methods of various parameters related to a schedulable object. They are self-explanatory.

2. Daemon status mutator and accessor methods which are used to query and to allow a handler to set/unset its daemon status, respectively.

3. Event handling methods that allows the programmer to create an application defined handler by subclassing the class and overriding the `handleAsyncEvent` and to manipulate the count of a handler. Note that, as metioned earlier, each handler has a count which is called `fireCount` of the number of outstanding occurrences. When an event occurs, the count is atomically incremented by one for each occurrence. The attached handlers are then released for execution under the condition that any minimum inter-arrival time constraint has been obeyed for sporadic handlers. `fireCount` is designed to remedy overload problems when handlers are being heavily used (i.e. event bursts - handlers that release before their previous releases have not been completed). In such cases, the fire count allows the currently associate server thread to continue the execution associated with the new release as soon as the old releases have finished. Therefore by correctly implementing and using the concept of `fireCount`, the associated overheads such as starting and stopping the server thread in which a handler run can be avoided. It is also possible to use `fireCount` aggressively by calling the protected accessor methods in the `AsyncEventHandler` class. Listing 2.6 demonstrates an aggressive approach to the use of `fireCount`. The `handleAsyncEvent` method in the `AsyncEventHandler` class is executed once and ignores all the event firings occurred between the first firing of the event and the call to the `getAndClearPendingFireCount` method. This is useful and efficient when it is sufficient to handle once for any number of the outstanding fire count. The `run` method is the one that will be called by its server thread when the handler is released. The method is `final` and therefore cannot be overridden. Therefore the `run` method of a handler will invoke its corresponding `handleAsyncEvent` method or the `run` method of the runnable parameter, repeatedly while the fire count is greater than zero.

As presented in Figure 2.3, the `BoundAsyncEventHandler` class extends `AsyncEvent-Handler` and most importantly has a dedicated server real-time thread (a server thread can be associated with only one bound handler for the lifetime of that handler)[9]. They

[9]Each `BoundAsyncEventHandler` has a dedicated real-time thread. The real-time thread can be implementation-dependent (i.e. kernel thread) or an instance of the `RealtimeThread` class.

```
// Aggressive approach to the use of fireCount
class AggressiveHandler extends AsyncEventHandler {
  public void handleAsyncEvent {
    // Do application-specific event handling here
    getAndClearPendingFireCount();
  }
}
```

Listing 2.6: The Usage of `fireCount` in the RTSJ

```
package javax.realtime;
public class BoundAsyncEventHandler extends AsyncEventHandler {
   /* constructors */
   public BoundAsyncEventHandler();
   public BoundAsyncEventHandler(SchedulingParameters scheduling,
                                 ReleaseParameters release,
                                 MemoryParameters memory,
                                 MemoryArea area,
                                 ProcessingGroupParameters group,
                                 boolean nonheap,
                                 Runnable logic);
}
```

Listing 2.7: The `BoundAsyncEventHandler` Class

are for use in situations where the added timeliness is worth the overhead of dedicating an individual real-time thread to each handler. The `BoundAsyncEventHandler` class is given in Listing 2.7.

In the RTSJ, happenings are referred to as external events and the software to handle happenings is contained in the classes `AsyncEvent` and `AsyncEventHandler`. The RTSJ specifies a three-layer mechanism for binding an external happening to an asynchronous event handler [16]:

- Interface layer: External happenings are platform-specific and are informed to the Java run-time of their occurrences. They can be operating systems signals on JVMs implemented on top of an operating system such as those defined by the POSIX standard, for example `SIGALRM` or hardware interrupts on JVMs that run without the help of an operating system. Each happening has a name which is chosen and assigned in an implementation-defined way.
- Asynchronous event layer: The `bindTo/unbindTo` method in the `AsyncEvent` class associates/disassociates an external happening with a particular event:

```
// Associates asyncEvent to Interrupt_5
asyncEvent.bindTo("Interrupt_5");
// Diassociates asyncEvent from SIGALRM
asyncEvent.unbindTo("Signal_A");
```

Listing 2.8: `bindTo/unbindTo` Methods in the `AsyncEvent` Class

The two methods in Listing 2.8, `bindTo/unbindTo`, tells the interface layer to direct/indirect the happenings with the specified names to/from the event object. The logic of the event object is concrete and the interface layer calls its `fire` method when the happenings occur. It then starts executing (or releasing) all the handlers and bound handlers that have been associated with it. For internally generated events, the `fire` method can be called directly by the application of JVM.

- Asynchronous event handler layer: `AsyncEventHandler` objects connect themselves to one or more `AsyncEvent` objects with the `addHandler` or `setHandler` methods in the `AsyncEvent` class. When an event is fired, it starts all the handlers attached to it. The event causes the `run` method in the associated handlers to execute. The `run` method calls the `handleAsyncEvent` method once for every time the event was fired (i.e. while `fireCount` is greater than 0. Recall that each handler has a count (called `fireCount`) of the number of outstanding occurrences. When an event is fired, the count is atomically incremented.). The `handleAsyncEvent` method is therefore to be overridden by subclasses to specify the application-specific code that gets executed when the associated `AsyncEvent` is fired.

As well as supporting external happenings (i.e. hardware interrupts and operating system signals), asynchronous event and their handlers can also be used to handle the following internal events:

1. Asynchronous error conditions detected by the RT-JVM such as a deadline miss or a cost overrun of a schedulable object: A schedulable object in the RTSJ can be associated with an `ReleaseParameters` object, which specifies the temporal requirements of the associated schedulable object. The following shows the most elaborate constructor for a `ReleaseParameters` object.

```
protected ReleaseParameters(RelativeTime cost,
                            RelativeTime deadline,
                            AsyncEventHandler overrunHandler,
                            AsyncEventHandler missHandler)
```

The cost refers to processing time units per release and the deadline refers to the time from a release that the schedulable object has to complete its execution. `overrunHandler` is an `AsyncEventHandler` object and will be called if an invocation of the associated schedulable object exceeds its cost on a particular release. Note that cost monitoring is an optional facility in an implementation of the RTSJ as this typically requires support from the underlying operating system. Similarly `missHandler` is also an `AsyncEventHandler` object and is invoked if the `run` method of the associated schedulable object is still executing after the deadline has passed. Null `missHandler` and `overrunHandler` values indicates that no handlers should be released when the deadline is missed or the cost overruns. With using asynchronous event handlers this way it is possible to deschedule the schedulable object that caused an error condition until some corrective action takes place and it is instructed to be rescheduled. As an aside, it is unclear why the RTSJ chose to use handlers for the error conditions rather than events.

2. Application-defined (software) error notification: Asynchronous events and event handlers in AEH offer an interesting alternative to faults and software events [16]. Traditionally Java programmers pass faults to other scopes by throwing an exception, returning an error code, or calling an error-handling method. Now it is possible to trigger asynchronous events if the system falls into unwanted situations as a general notification or a fault-handling mechanism.

3. Time-triggering events: Event handlers can also be time triggered. The abstract `Timer` class in the RTSJ defines the base class from which timer events can be generated. There are two subclasses of the `Timer` class, `OneShotTimer` and `PeriodicTimer`. The `OneShotTimer` class represents a timer that will only fire once (unless it is restarted). The `PeriodicTimer` class encapsulates a timer that will fire on the first occasion at the time indicated by the start time parameter in its constructor method and periodically according to the time indicated by the interval time parameter in the constructor.

As shown in this subsection, the RTSJ has made a good attempt to generalise real-time activities away from real-time threads towards the notion of schedulable objects, and intends to provide a framework from within which priority-based scheduling with online-feasibility analysis can be supported [70].

2.3.2 Configurability and Implementation of AEH

As discussed in Chapter 1, the RTSJ neither facilitates a configurable framework to finely tune its AEH subsystem to fit the application's particular needs [67], nor provides any implementation guidelines in order to allow variation on implementation decisions [5]. Whilst the goals of asynchronous event handling in the RTSJ are laudable, their realisation in the current version of the specification suffer from the following limitations due to the lack of comprehensive configurability:

- A single model for all types of events handlers — all asynchronous event must be handled in the same implementation-dependent way; the `AsyncEventHandler` class provides the concrete implementation of an AEH algorithm that mandates the way of handling events and it is not possible for an application to indicate a different implementation strategy for various handlers with different characteristics such as interrupt handlers or non-blocking handlers.

- Lack of implementation configurability — the `AsyncEventHandler` class provides no facilities for the programmer to configure its AEH and therefore the application is unable to finely tune its AEH components such as the thread pool or the AEH mapping algorithm [35].

The above problems in the RTSJ severely limit the flexibility of the AEH implementation and consequently the RTSJ's model has been criticised as lacking in this area [35, 46, 69]. This also makes achieving the lightweightness of AEH entirely dependent on the implementation.

The driving design goal of the AEH model of the RTSJ is to have a lightweight concurrency mechanism that is capable of support hundreds (if not thousands) of handlers; consequently handlers must not incur the same overheads as real-time threads. Therefore, if a system needs many `AsyncEventHandlers` and the binding latency is not critical, consider minimising the use of `BoundAsyncEventHandler` as they incur the same overhead as real-time threads, meaning that they are not lightweight. The lightweightness can only be achieved by using as few as possible server threads to execute as many as possible handlers. Doing this, results in smaller data structures in the real-time virtual machine, less stack space for server threads, fewer context switches between server threads etc. The trade-off is an increase in the latency between the event being fired and a handler being assigned to a server thread for execution, compared to the static 1:1 relationship used for bound asynchronous event handlers.

Some implementation will, however, simply map an event handler to a real-time thread at run-time and the original motivations for event handlers will be lost as they become as heavy-weight as threads. The point here is that enabling event handlers to use far fewer system resources than actual real-time threads do, and therefore not to suffer the same overhead as them is a primary goal of the RTSJ. Therefore, the key challenge in implementing asynchronous event handlers is to limit the number of real-time threads without jeopardising the schedulability of the overall system. Furthermore, the resulting implementations must provide efficient and predictable mappings between these two entities, the asynchronous event handler and the real-time thread.

In [16, 69], possible implementation strategies for asynchronous event handling have been proposed and the feasibility analysis, if one server thread is used, also has been derived. The single server approach is one extreme of the possible implementation strategies whilst dynamically allocating a distinct server thread for each handler resides at the other extreme. Although the latter is simple to implement and adequate for systems with a small number of events, it is expensive for a large number of events and therefore do not achieve the primary goal of the RTSJ. With a single server approach it is recommended that the order of the handler queue should be priority-ordered to enable effective schedulability analysis. The priority of the server thread dynamically changes to reflect the priority of the handler it is executing. Furthermore it is necessary to implement a priority inheritance algorithm in order to get usable bounds for the priority inversion, as a released higher priority handler will not immediately be executed even if a lower priority handler is executing. The priority of the server should 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 head of the handler queue. The time at which the priority of the server increases is based on the inheritance algorithm in use (i.e. basic inheritance and priority ceiling protocol). Although the priority of the server changes in this approach, it is essentially a fixed priority system. Two main disadvantages with the single server approach are also identified as follows:

- Potentially Unbounded Priority Inversion - A priority inversion occurs whenever a lower priority task executes while some ready to execute higher priority task waits [45]. With the single server approach, priority inheritance algorithms [6] give usable bounds when the server causes priority inversion. However it only works under the assumption that the currently executing handler does not block (e.g. issue a sleep or wait method call) as higher priority pending handlers will also be indefinitely delayed. Multiple servers are required to avoid this problem.

- No-Heap Event Handlers - asynchronous event handlers in the RTSJ may indicate that they will not access heap memory to safely preempt the garbage collector. The single server model will fail when there is a mixture of heap and no-heap event handlers. The solution to this problem is again to have two set of multiple servers, heap and no-heap, and two handler queues. The set of heap servers executes heap event handlers and the other set of no-heap servers performs no-heap event handlers.

The standard response time equation for an arbitrary task i in a fixed-priority system, where the response time of tasks are smaller than or equal to their respective periods, is given below [2]:

$$R_i = C_i + B_i + \sum_{j \in hep(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \tag{2.1}$$

where R_i, C_i, B_i, and T_i represent the worst-case response time, the worst-case computation time, the worst-case blocking time[10] and the period of thread i, respectively. Also $hep(i)$ represents the set of tasks with a higher or equal priority than/to i. The equation starts from $R_i^1 = C_i + B_i + \sum_{j \in hep(i)} C_j$, and computes iteratively until $R_i^{l+1} = R_i^l$ for some $l \geq 1$. The above equation is extended to capture the single server model as the following [69]:

$$R_i = C_i + B_i + \sum_{j \in hept(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j + \sum_{l \in heph(i)} \left\lceil \frac{R_i}{T_l} \right\rceil C_l + \max_{k \in leph(i)} C_k \tag{2.2}$$

In Equation 2.1, the interference from only the threads with a higher priority is considered. However Equation 2.2 takes the non-preemptible nature of the single server approach into account by incorporating the maximum of all low priority event handlers. $hept(i)$ represents the set of threads whose priority is higher than or equal to thread/handler i and

[10] B_i represents the maximum blocking time that thread i can suffer for the priority ceiling protocol model currently in use for the system. In this thesis, the term blocking (or self-suspending) is also used to designate the time during which a thread is suspended for some external operation, such as an I/O operation or a remote procedure. Therefore B_i used here differs from the blocking (or self-suspending) time of thread. The schedulability analysis that includes the blocking factor is discussed in detail in Section 4.5.

$heph(i)$ illustrates the set of handlers whose priority is higher than or equal to thread/hander i. $leph(i)$ is the set of handlers whose priority is lower than or equal to thread/handler i. The worst-case non-preemptible time is therefore the maximum of all the low priority handlers.

Another implementation strategy in both papers is to use a priority-ordered queue (or one queue per priority level) and multiple server threads. This approach is more flexible than a single server approach in terms of dealing with blocking event handlers. However it is still required to create a server when all the servers in the system block to avoid unbounded priority inversion and assume that there may be some contention for shared software resources (i.e. handler queue). The dynamic creation and protecting shared resources however are relatively expensive in terms of both time and space. Also there are other implementation issues to address, such as defining the points where and when a server changes its priority to reflect the priority of the handler which the server is currently executing, and clarifying the circumstances under which a server is allowed to continue executing another handler after completing the execution of the current handler. Without having these issues addressed, it is likely that the model causes unnecessary context switches or leads to invocation of superfluous servers.

In this section, only sporadic and periodic asynchronous event handlers are considered which is the main focus of this thesis. However, to have analysable aperiodic event handlers, it is required for them to use an aperiodic server technique [71]. This is because aperiodic schedulable objects have no well-defined release characteristics and therefore they may impose unbounded demand on the processor's time [6]. The one that is most relevant to the RTSJ is a deferrable server using processing group parameters [67]. When processing group parameters are assigned to one or more aperiodic schedulable objects, a server thread is effectively created with its start time, cost (capacity) and period which collectively define the points in time where the server's capacity is replenished. Any aperiodic schedulable object that belongs to a processing group is executed at its defined priority while the server still has capacity. When capacity is exhausted, the aperiodic schedulable objects are not allowed to execute until the start of the next release when its capacity is replenished. If the application only assigns aperiodic schedulable objects of the same priority to a single processing group, then the functionality of a deferrable server can be obtained.

2.4 Summary

This chapter has outlined the research that is relevant to the design and implementation of asynchronous event handling for real-time systems. We began with an overview of concurrent programming models, emphasising the flexibility of threads and the efficiency of events. As both of them has its own benefits when properly implemented, it is concluded that the mixed programming model should be supported in order to obtain the benefits from both models such as high throughput, ease of facilitating parallelism and programming. This ultimately allows the program to tailor the application to fit its specific needs.

A survey of event handling facilities provided by main stream programming languages and middleware systems was also presented. Either directly or indirectly they support a notion of threads and a level of asynchrony to facilitate concurrent programming. Some programming languages such as C# do not provide asynchronous event handling facilities by its own semantics and specification. Such languages rely on the underlying operating system (i.e. POSIX) or the library of pre-coded APIs (i.e. .NET framework). As mentioned earlier, the section, event handling in distributed systems, mainly discusses a higher level of abstraction for communication between distributed entities. This differs from event handling approaches in centralised systems discussed in the previous section, although there are some similarities (e.g. their classification is based on the synchronisation decoupling). However delivered messages or events must be handled using a certain event handling technique in the receiver's perspective. Communication models used in some well-known distributed systems are presented first and then an exemplary event handling implementation for distributed systems was discussed with respect to its local event handling approach. Regardless of their internal and external structures they can be classified into two class with respect to the mapping between the executionees and the executioners. One is the single server approach (i.e. where the single server thread executes all the released handlers) and the other is the 1:1 mapping (i.e. where a single server thread executes a single handler). Some of the investigated languages or middleware systems are able to support both forms. However there is no framework that provides asynchronous event handling with fewer server threads than the number of handlers concurrently active.

Lastly, asynchronous event handling in the RTSJ was discussed in terms of its basic API model and implementation. In the RTSJ, an occurrence of an events may be initiated by application logic, by mechanisms internal to the RTSJ implementations (i.e. handlers with periodic release parameters), or by the triggering of a happening external to the JVM (such as a software signal or a hardware interrupt handler). An event can also be fired in program logic by the invocation of the `fire` method of an event. The triggering of an event due to a happening is implementation-dependent except as specified in `POSIXSignalHandler` which lets the programmer handle POSIX signals if the underlying operating system is compliant to POSIX. An event can have a set of handlers associated with it, and when the event occurs, the `fireCount` of each handler is incremented, and the handlers are released. An event is an object that represents an internal or external happening and is used to inform event occurrences to handlers. The relationship between events and event handlers is many to many and handlers can be associated with a POSIX signal. The RTSJ allows considerable freedom in the implementation of asynchronous event handling. We have also considered two prevailing models for asynchronous event handling from the single server approach to the simple thread per handler model, found in the literature. This chapter was concluded by identifying the outstanding issues associated with AEH in the RTSJ, efficiency and flexibility.

Chapter 3

Evaluating AEH Models in Current RTSJ Implementations

In appraising the efficiency and scalability of asynchronous event handling models for real-time Java applications, this chapter presents and discusses various asynchronous event handling algorithms used in some popular RTSJ implementations. To achieve the lightweightness requirement of AEH in the RTSJ, a real-time server thread should execute as many released handlers as possible without causing any concurrency-related issue. Therefore it is especially highlighted how the mapping between asynchronous event handlers and real-time server threads is performed in the RTSJ implementations to reduce the number of real-time server threads required.

In Section 3.1 various asynchronous event handling algorithms are investigated regarding their respective mapping approaches. In Section 3.2 two exemplary RTSJ implementations, jRate and Java RTS, are designed and constructed using the networks of automata formalism provided in the UPPAAL model checking tool. The constructed AEH models give us interactive analysis of their run-time behaviour during early design stage and also offers the exhaustive dynamic behaviour for proving safety and liveness properties. Section 3.3 presents the formal analysis of the two constructed models by simulating and verifying them using CTL (Computation Tree Logic) [4] to guarantee that they are free from unwanted behaviour and to examine their respective least upper bounds of real-time server threads.

3.1 Current AEH Implementations of the RTSJ

There are several different approaches to implementing asynchronous event handling (AEH) in the RTSJ. This section presents the approaches used in some popular RTSJ implementations. The descriptions for the AEH implementations presented here give us their general run-time behaviour and explain the respective pros and cons of their approaches. The RTSJ anticipates future embedded systems having a large number (even hundreds of thousands [5]) of handlers. By limiting the number of concurrently active server threads for the execution of released (or pending) handlers, the system can avoid throughput degradation and therefore the overall performance will be higher than that of an unconstrained server-per-handler model. Consequently this will enable the system to be more efficient (i.e. lightweight) and scalable (i.e. more handlers). It is, therefore, important to appreciate how the AEH algorithms are implemented to deal with multiple handlers simultaneously released in the system. The implementations presented below include the Reference Implementation (RI), OVM, Jamaica, jRate, and Java RTS. It should be noted that the only open-source RTSJ implementation available allowing its source code to be accessed is jRate [12]. The detailed behaviour of AEH in other implementations presented here is obtained via private communications [22, 29, 50] not via formal documentation. The following defines the classification of AEH mapping models based on the number of handler(s) that a server thread executes during a single release at run-time:

- 1:1 Mapping - An invoked server thread executes one handler and goes back to the waiting state where it sleeps until invoked again. The mapping can be either *static* or *dynamic* as follows:
  - Static - Each server thread has a statically bounded handler. In other words, handlers are permanently associated with a server thread and the server thread can only be dedicated to a single bound handler for execution at any one time. The static 1:1 mapping model must be used to implement bound asynchronous event handlers [5]. The OVM [29] implementation of the RTSJ uses this approach for all handlers.
  - Dynamic - The bindings between server threads and handlers take place at run-time dynamically. There is, therefore, inevitably some latency between the handler being released and the server thread starting to execute the handler. Unlike the static 1:1 mapping model, server threads are chosen to be invoked at run-time (i.e. servers are not statically bound to handlers, providing full

space decoupling of both sides). The dynamic 1:1 mapping model is used in the RTSJ's reference implementation (RI) [16] and jRate [12].

- 1:N Mapping - A 1:N mapping implies that N handlers are executed by a single server thread. The invoked server thread executes N handlers during a single release that is between the instants from being invoked to relinquishing the CPU. The mapping can also be either static or dynamic as follows:

  · Static - A server thread has a statically bound group of handlers to execute. In other words, the group of handlers are permanently associated with a server thread and the server thread can only be dedicated to the group for execution at any one time. N can therefore be in the range from 1 to the number of handlers in the group. The static 1:N mapping model is used in Jamaica [22] by having a dedicated server thread per priority-level.

  · Dynamic - An invoked server thread executes N handlers per invocation, where N is in the range $[1,\infty]$. Once the server thread is invoked, it executes N handlers in turn before going back to the waiting state. The actual value N varies, depending on the AEH algorithm and the event firing sequence. Here again, unlike the static 1:N mapping model, server threads are chosen to be invoked at run-time (i.e. servers are not statically bound to handlers, providing full space decoupling of both sides). Java RTS 2.0 [50] tries to employ the dynamic 1:N mapping model. However there is a phenomenon called the MSSP, that prevents the implementation from fully realising this mapping model. For more information about the phenomenon, see Section 3.1.5 and 3.3.4.

3.1.1 The Reference Implementation (RI) from TimeSys

The RI [60] uses a simple scheme for AEH. It creates a server thread each time a handler is released and there is no active server thread for that handler [16]. The created server thread terminates when the fire count for that handler goes to 0 (i.e. all outstanding firings of the event have been handled). This AEH implementation is the most intuitive way of constructing the AEH subsystem. The mapping of server threads to handlers in the RI can therefore be classified as a dynamic 1:1 mapping, whilst a static 1:1 mapping is referred to that is used for bound handlers that provide a dedicated server thread per handler for their life time. In addition to the run-time server creation and destruction overhead, the dynamic 1:1 mapping will also likely produce a proliferation of server threads along

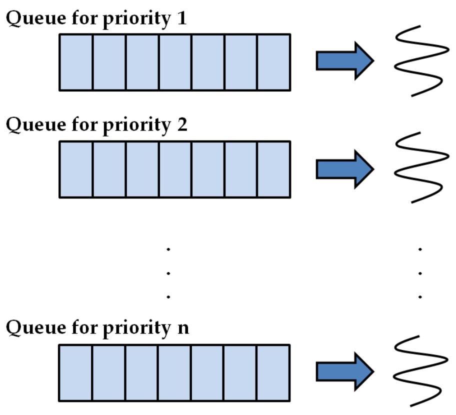


Figure 3.1: AEH in Jamaica

with per thread overhead and context-switches when released handlers are many and their control algorithm is simple and non-blocking.

3.1.2 OVM from Purdue University

In OVM [29], each handler has a dedicated server thread permanently bound to it for the life-time of the handler. Consequently the AEH implementation of the OVM is not different from the expected implementation of bound handlers of the RTSJ. It therefore does not realise the main motivation of having non-bound handlers in the RTSJ.

3.1.3 Jamaica from Aicas

The AEH implementation in Jamaica [22] allocates a dedicated server thread per priority level. Each priority level has a queue and a dedicated server thread. Therefore it realises a static 1:N mapping in the absence of dynamic priority changes, as depicted in Figure 3.1. This works well as long as the handlers' logic is non-blocking. The least upper bound of server threads used in Jamaica is smaller than that of the static or dynamic 1:1 mapping. The priorities of server threads are static and therefore the run-time priority changes of server threads are not required. This property of Jamaica considerably reduces the complexity of its AEH implementation. However it causes unbounded priority inversion if a handler blocks. When a server thread executes a handler and it blocks, a lower priority server thread is given the CPU and executes its handler. If the blocked server thread has pending handlers in its associated queue, they are also blocked until the corresponding server thread unblocks. To avoid this, it is necessary to dynamically create a new server thread (or take one from a pool of server threads if available). However the implementation

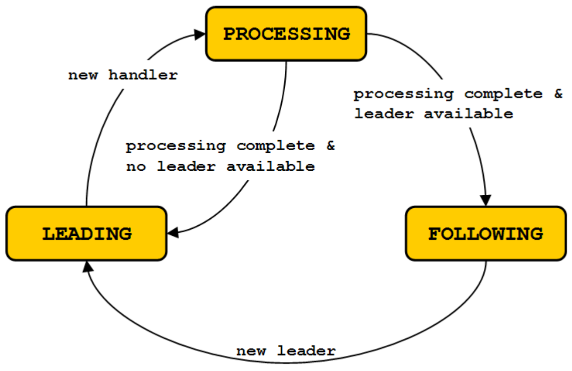

PROCESSING
new handler
processing complete & leader available
processing complete & no leader available
LEADING
FOLLOWING
new leader

Figure 3.2: States and Valid Transitions in the Leader/Followers Pattern

does not take the corrective action when required.

3.1.4 jRate from Washington University

The AEH implementation in jRate [12] uses a well-known architectural design pattern, the *Leader/Followers* [55][1], that provides a concurrency model where multiple server threads can demultiplex handlers and execute them. The main component of the Leader/Followers design pattern is a pool of server threads by which the released handlers are executed. There is only one active server thread at a time, the *leader*, that waits for a handler to be released on a set of events. Meanwhile, other threads, the *followers*, can queue up waiting their turn to become the leader. After the current leader thread detects a handler, it first promotes a follower thread to become the new leader and then it plays the role of a processing thread, which demultiplexes the event handler and perform application-specific event processing. Multiple processing threads can run concurrently while the current leader thread waits for a new event handler to arrive. After executing its handler, a processing server thread reverts to a follower that waits in the pool for its turn to become the leader thread again. Having the pool of server threads, run-time server creation and destruction overhead is avoided and the server threads in the pool take turns de-multiplexing handlers. Figure 3.2 illustrates the states and the valid transitions of threads in the Leader/Followers pattern.

A UML sequence diagram for jRate AEH is shown in Figure 3.3. The essential behaviour of the AEH implementation in jRate is based on the Leader/Followers design pattern described above. However it simplifies some of the actions performed in the de-

[1]This design pattern is also used in the TAO research project presented in Section 2.2.4.

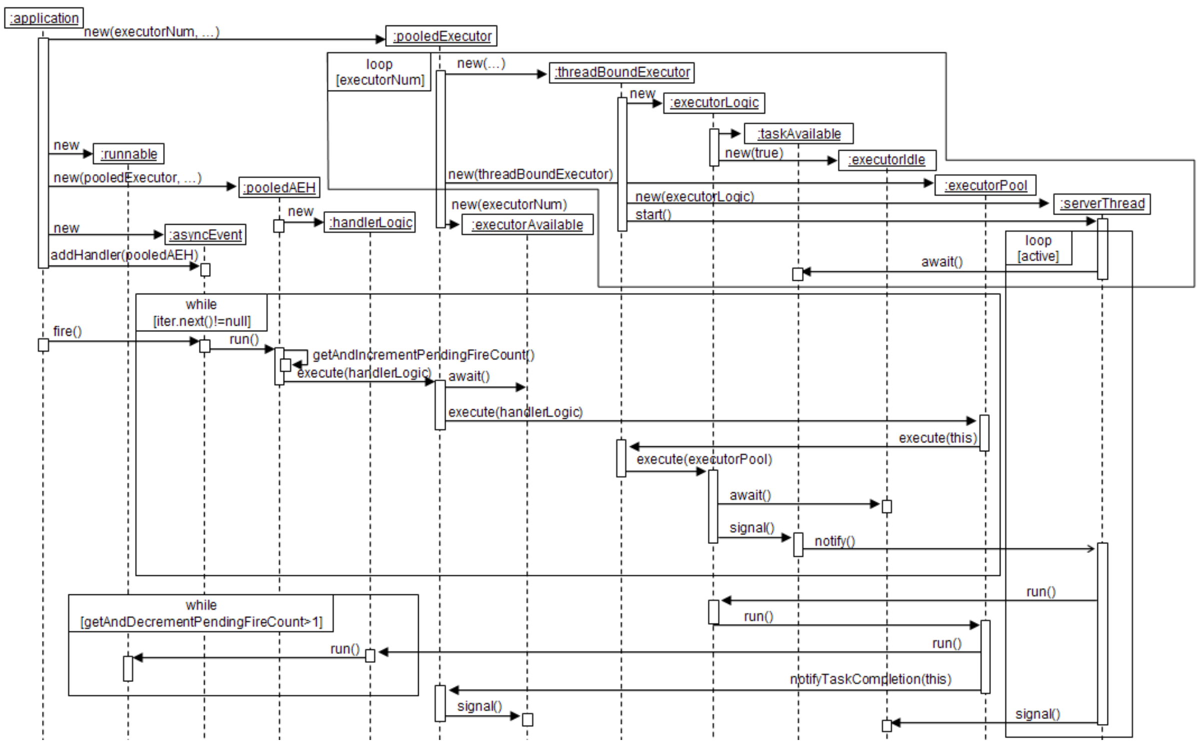


Figure 3.3: UML Sequence Diagram for jRate AEH

sign pattern; the leader does not promote a follower to be the new leader but the run-time or event firing thread will notify a follower directly to be the new leader. In order for handlers in jRate to be executed by a pre-defined set of server threads, the programmer has to create an instance of the `PooledExecutor` class and pass it to handlers as a parameter when they are constructed. Then the instance of `PooledExecutor` plays a major role as a server pool in the implementation and manages key AEH structural components such as server threads, event flags, and a counting semaphore. When a server pool is constructed, the number of server threads to be created in the pool can be specified via an integer variable, `executorNum`. Other schedulable related parameters can also be passed in at the construction of server pools. There are two event flags, `taskAvailable` and `executorIdle`, created for each server thread. When a server thread is started, it waits on `taskAvailable` until notified by the run-time or the event firing thread. `executorIdle` is used to block the run-time or the event firing thread while its associated server thread executes. `executorAvailable` is a counting semaphore that is a counter for a set of available resources, rather than a locked/unlocked flag of a single resource. It therefore holds the number of available server threads and decides to continue or block the execution based on the availability of the server threads. When a server thread is notified and therefore becomes active, the application-specific event handling is performed via the `run()` or `handleAsyncEvent()` methods in the currently associated handler. The handler is executed while the number of `fireCount` is greater than 0. After the completion of the handler the server signals the counting semaphore `taskAvailable` to make itself available for future use and also signals the `executorIdle` event flag to indicate the server is idle.

It is therefore easy to see that the mapping between the server thread and the handler in jRate AEH is a dynamic 1:1 mapping based on the Leader/Followers design pattern. The run-time or the event firing thread notifies an available server thread when a handler is released. The number of server threads required to execute a certain number of released handlers is discussed later in this chapter using the UPPAAL model checking tool. There are two other main drawbacks in the jRate AEH implementation. One is that the run-time or the event firing thread blocks when all the server threads in the pool are either ready or running until one of them becomes idle. This inevitably incurs unbounded priority inversion when a handler with a higher priority is released and all the server threads in the pool are busy executing lower priority handlers. The other shortcoming is that handlers with different priorities are executed at the server threads' priority level in a FIFO manner as if they were assigned the server threads' priority. This is due to the fact that the server threads in jRate do not change their priority levels to reflect that of the handlers that they are currently executing and all the server threads in the pool will have the same level of priority that was assigned when the pool is constructed. If the released handler has a higher priority level than that of the server threads, then another form of priority inversion will occur. Therefore programmers may not assume that handlers will be executed according to the priorities that they were originally assigned. Although this static priority approach significantly reduces the complexity of the AEH implementation in jRate, it is obviously inadequate for real-time systems due to the priority inversion. A possible solution to this problem would be to create an instance of `PooledExecutor` for each priority level and to assign handlers to the corresponding `PooledExecutor` with the same priority. However this will incur considerable overhead in terms of both time and space when priority levels are many.

3.1.5 Java RTS (Real-Time System) from Sun Microsystems

The Java RTS AEH 2.0 implementation [50] consists of two main data structures, queues of pending handlers (i.e. this refers to queues which store handlers that have been released, but not started their execution yet) and a pool of server threads waiting to be notified (or invoked) to execute pending handlers. The queues of pending handlers are used to store and dispatch the released handlers in a priority ordered manner. The server pool is used to manage a set of server threads by which the pending handlers in the queues are executed. The programmer can configure the size of the pool and additional server threads may also be created when a handler is released and all server threads in the pool are busy.

The run-time behaviour of Java RTS AEH is quite different from the other implementations and exhibits two particularities in order to enable asynchronous event handlers in Java RTS to be more lightweight, in the effort to realise the dynamic 1:N mapping model. The implementation tries to achieve this by allowing server threads to execute more than one handler as long as it does not produce any concurrency related side-effect such as deadlock and priority-inversion. One of the particularities of the Java RTS AEH implementation is that server threads changes their priorities to reflect those of handlers that they are executing at the time, and the binding between the released handler and the server thread is dynamically performed at run-time when the handler is scheduled for execution, not at its release time, *a late binding technique*. The main idea is to maintain a server thread ready as soon as a handler is released. When a handler is released, it is enqueued in the pending handler queue at its priority as there is one queue per priority level. After the insertion, the run-time or the event firing thread which released the handler looks for the *ready server*. In the RTSJ 'ready' means a server that is in the 'ELIGIBLE-FOR-EXECUTION' state - a `RealtimeThread` can be (i.e. EXECUTING, BLOCKED, ELIGIBLE-FOR-EXECUTION). However, in the context of the Java RTS AEH a '*ready server*' requires that the server is ready but not preempted. This is because a preempted server already has a handler bound to it, and it must not be allowed to change its priority as it has already been assigned the right priority level for the handler being executed by the server. If there is the ready server and its priority is higher than or equal to that of the released handler, no further action is required. If the ready server has a priority lower than the released handler's priority, then the ready server's priority is increased to the handler's priority level. If no ready server exists, a new one in the pool is notified (or one is created, if no server threads available at the time of the release), its priority is configured accordingly, and becomes the new ready server.

Another particularity of the Java RTS AEH implementation is that the current running server notifies a server thread in the pool right before it starts to execute the handler that it removed from the queue, only if there are one or more pending handlers. When a ready server thread gets the CPU (its state changes from `ELIGIBLE-FOR-EXECUTION` to `EXECUTING`), it picks up the first handler from the pending handler queue corresponding to its priority or it will look for a pending handlers in queues of lower priorities if the corresponding queue at its priority is empty. If it does not find any pending handler, it returns to the thread pool and waits until it is picked up from the pool again: the server thread is now no longer *ready*. If the server thread finds a pending handler, it removes the handler from the queue and it rechecks whether the queue is empty (and also lower priority queues if the queue at its priority is empty) after removing the handler. If the server thread finds another pending handler, it notifies a server thread in the pool or creates a new server thread and the newly notified (or created) server thread becomes ready for the priority of the first found handler. If all queues are empty the server thread starts to execute the handler without notifying or creating a new server thread. Finally the current server binds itself to the handler that it has removed from the queue, set up the execution context according to the handler's timing parameters and starts executing the application-specific event handling. If the handler contains blocking code, the ready server thread will execute the remaining pending handler(s). If a higher priority handler is released at this time, the run-time or the event-firing thread will increase the priority of the ready server thread to that of the released handler. The ready server thread therefore will preempt the running server to prevent priority inversion. When the server thread finishes the execution of the handler, it checks the queue for pending handlers. If there are, the server will bind itself to the highest priority handler in the queue and executes it after adjusting its priority accordingly. Otherwise, the server returns to the pool. Before the execution of the handler, the server notifies one of the server threads in the pool to have a new ready server at the priority of the first handler in the queue only if there are pending handlers in the queue as done before. Therefore when the current running server blocks, the ready server is going to execute the pending handlers. It should be noted that the server thread in Java RTS AEH never increases its priority by itself except when it is ready.

The following simple scenario that is taken from the private communications with Sun Microsystems for Java RTS 2.0 AEH algorithm [50] explains its run-time behaviour: "When a server thread `T1` in the pool is invoked to execute a handler, it does not promote another thread `T2` to be ready right after it is active. It promotes another server thread to be ready right before it starts executing the handler it just has taken out from the handler queue and therefore thread `T2` will continue the job and will take care of the next released handler if `T1` blocks. When `T1` completes the execution of its handler, it does not return immediately to the pool. First, it checks if another handler is released and not bound yet. If it finds one, it will bind itself to this handler and will execute it. It can bind itself to a handler running at its current priority or find a handler with a lower priority, in this case its priority is automatically adjusted during the binding process. With this mechanism and the run-to-block semantic of the real-time priorities, thread `T1` is able to serve a number of handlers without additional context switches (if none of the handlers includes blocking code). When no more released handlers are available in the queue, thread `T1` finally returns itself to the pool. Thread `T2` will take the CPU, find no pending handlers and returns itself to the pool too."

3.2 Formal Automata Models in UPPAAL

UPPAAL [61] is a toolbox for modelling (via graphical editor), validation (via graphical simulation, simulator) and verification (via automatic model-checking, verifier) of real-time systems jointly developed by Uppsala and Aalborg University. UPPAAL systems are modelled as networks of timed automata extended with integer variables, structured data types, and channel synchronisation. The simulator is used for interactive analysis of system behaviour during early design stages while the verifier, which is a model-checker, covers the exhaustive dynamic behaviour of the system for proving safety and bounded liveness properties. The verifier, which is a symbolic model checker, is implemented using sophisticated constraint-solving techniques where efficiency and optimisation are emphasised. In this section, UPPAAL (version 4.0.10) is extensively used to design, simulate, and verify the AEH implementations as it offers the following advantages:

- It offers a user-friendly environment for modelling target systems via graphical editor.
- Verification is based on a relatively well-known language, CTL. Therefore it is easy to learn and understand.
- UPPAAL has been successfully applied in many case studies from communication protocols to multimedia applications [32].
- It also incorporates the notion of time.

Other than the above, the most important property that UPPAAL offers for designing an AEH model is that it provides an inexpensive means for interactive analysis of the designed system's behaviour during early design stages. Other widely-used model checking tools such as SPIN [42] do not provide interactive analysis of the system's behaviour during early design stage. Their simulation and verification are normally based on a complete specification. This particularly provides an important benefit when designing an AEH model which does not have a concrete and complete specification. For example the tool allows many different schemes for notifying server threads and relinquishing the CPU to be employed during early design stage so that their respective partial behaviour can be observed at run-time before defining any concrete specification. Based on this observation, more efficient schemes can be found.

This section presents the AEH subsystems of two RTSJ implementations, jRate and Java RTS, designed using the networks of automata[2] (Readers are referred to [4] for a more detailed explanation about UPPAAL). These two implementations are chosen as they use relatively advanced AEH algorithms and hence are non-trivial. For example jRate uses the Leader/Followers design pattern for its AEH and Java RTS uses the late binding technique in the effort of reducing the number of server threads required at run-time.

3.2.1 Modelling Architecture

This section presents a modelling architecture to design an AEH implementation using the automata formalism of the UPPAAL tool. The idea of the architecture is to decompose the functionalities of the AEH system into several different conceptual components. The components included in the models are the environment, handler queue, thread pool, and base scheduler. They represent the atomic building blocks for constructing the AEH subsystem and allow their run-time status to be observed. For example the handler queue hold the information regarding the number of pending handlers and their respective priority levels and the information is used by other components at run-time to simulate a certain AEH mapping. This applies to other components as well. They provide the essential information for the AEH subsystem to work correctly. The base scheduler actively controls the overall behaviour of the AEH subsystem using the provided information (i.e. scheduling server threads). Some components which exist in the actual implementations have been abstracted away. Such components include the application and thread pool management

[2]The source for all the UPPAAL automata models presented in this thesis is freely available at `www.cs.york.ac.uk/rts/djmin/work/AutomataModelsForAEHtecs.zip`. It also includes all the queries used to verify the models in this thesis. The queries that verify safety and liveness properties are presented in Section 3.3. Note that simple reachability properties are not presented in this thesis, but are included in the source.

layer. It is valid abstraction due to the following reasons. First, the application responds to a fired event in the environment by releasing the associated handlers. However, from the AEH subsystem's perspective the results are the same with the environment releasing and queuing the associated handlers directly. Second, the thread pool management represents the operations, which controls the invocation and synchronisation of waiting server threads in the thread pool, to execute the pending handlers. In the designed AEH models in this thesis, the thread pool component controls itself, managing the invocation and synchronisation of waiting server threads. Therefore such components that can be abstracted away have been omitted from the models to reduce the size of the state space to speed up the verification. All the components are designed into two automata in each model and are presented below:

- Environment: This component represents the environment that non-deterministically fires an event which, in turn, releases an associated handler. The priority levels of released handlers can be varied based on the simulation and verification scenarios. Note that in the RTSJ every occurrence of an event increments `fireCount` by one in each attached handler, and an event may have one or more associated handlers. To reduce the complexity of the modelling we assume that `fireCount` never goes beyond 1 (i.e. the same event releases a different handler each time it is fired) and an event has only one associated handler. The former assumption is valid in the sense that once a server thread is bound to a handler it will be used to execute the handler while `fireCount` of the handler is greater than zero and therefore the mapping will stay the same. The latter assumption is also valid in the sense that events in the model can be fired randomly without timing restrictions allowing the model to have the full randomness and hence firing events twice or more consecutively has the same effect as an event releasing two or more handlers.

- Handler Queue: This component models a pending handler queue and the operational methods on the queue for adding and removing handlers in a priority-ordered manner. The AEH implementation in jRate does not use this component as it dispatches released handlers immediately to available server threads without temporarily holding them in a queue.

- Server Thread: This component is responsible for the representation of four states that server threads can hold at any given time, namely waiting, ready, running, preempted. When server threads are created they are in the waiting state and are made ready upon the occurrence of handlers' being released. Ready server threads are scheduled for execution and are made running based on their priorities. When they are in the running state, a server thread may be preempted when there is a higher priority server thread waiting in the thread run queue. The blocked state is absent in the component in order for them to be given a well-defined cycle of execution. If server threads are allowed to block, the model will indefinitely or frequently block, resulting in memory exhaustion when verifying the models. Furthermore it is easy to see that if a handler contains blocking (or self-suspending) code, a separate server thread is required to prevent priority inversion. For example if a server thread blocks while executing a handler, all the pending handlers will also block (not being able to execute until the blocked server unblocks). To prevent this it requires another server to continue executing the pending handlers. If that server blocks as well, it needs another server thread again. This iteration lasts until there are no pending handlers, resulting the system to require the same number of server threads as the number of blocking handlers[3].

- Thread Pool: This component represents a pool of server threads. This component is responsible for managing server threads preallocated in the pool at the initialisation phase. The number of server threads can typically be configured when started. They are notified when handlers are released and dynamically join the pool to wait for a new released handler when they complete.

- Base Scheduler: This component represents the basic functions and interactions of the base scheduler, `PriorityScheduler`, of the RTSJ. The scheduler schedules server threads preemptively in a priority-ordered manner and allows a server thread to run, yield, and be preempted. It checks the current status of the AEH system at normal scheduling points (i.e. when a handler is released) and is responsible for activating and deactivating server threads in the pool accordingly.

The above components are combined and accordingly modified to realise each AEH implementation. Some component may even be absent from a certain AEH model and some are extensively modified to fit the actual AEH algorithm used in the implementations. However it should be noted that the base abstraction template constructed and used for the AEH automata models in this thesis is the same in order for different AEH automata models to be fairly compared.

[3]This proposition is discussed in detail in Section 4.1.

3.2.2 AEH Automata Models in UPPAAL

In this section we presents the AEH automata models for the two AEH systems used in *jRate* and *Java RTS*. The models have been designed with a number of modelling patterns recommended in [4] to speed up the verification and reduce the size of the state space (i.e. the overall memory consumption). The design patterns used in the models include variable reduction, synchronous value passing, atomicity, and abstraction.

```
const int PRIORITY_LEVELS = 6; // priority levels for handlers
const int MAXFC           = 6; // maximum number of released handlers

// selection variable for priorities
typedef int[1, PRIORITY_LEVELS] p_levels;
// selection variable for threads
typedef int[1, MAXFC]           threads;

// Four possible states for server threads in the models
const int WAITING   = 0;
const int READY     = 1;
const int RUNNING   = 2;
const int PREEMPTED = 3;

int  overallSTsRequired;        // overall server threads required
int  overallSTsInvoked;         // overall server threads invoked
int  overallHandlersReleased;   // overall handlers released
int  invoked[threads];          // number of a server thread invoked
int  CS;                        // context-switches
int  st_stat[threads];          // states of a server thread
bool procBusy    = false;       // is the CPU busy?
int  p_th        = 0;           // priority of current running ST
int  t_id        = 0;           // id of current running ST

// Synchronisation Channels
chan notify[threads],
     run[threads],
     yieldCPU[threads],
     preempted[threads];

// Returns the first handler in the pending handler queue
int phqFirst();

// Returns the second handler in the pending handler queue
int phqSecond();

// Returns true if and only if the pending handler queue is empty
boolean isphqEmpty();

// Insert a handler in the pending handler queue according to
// its priority
void putHandler(int pri);
```

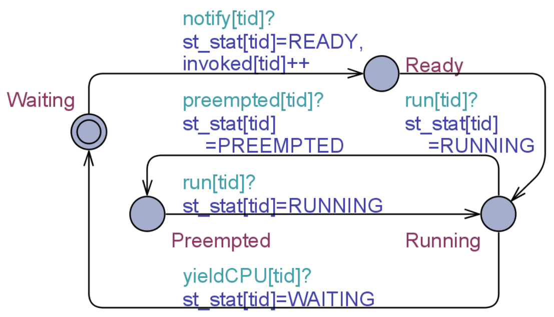


Figure 3.4: `ServerThread` Automaton for jRate AEH

```

// Remove the first handler from the pending handler queue
void getHandler();

// Notify a server in the thread pool
int notifyServer();

// Returns the first server in the thread run queue
int trqFirst();

// Returns true if and only if the thread run queue is empty
boolean istrqEmpty();

// Insert a ready server into the trq in a priority-ordered manner.
void putThread(int pri, int thread);

// Insert a preempted server into the trq in a priority-ordered
// manner. It will be added to the front of all server threads
// with the same priority.
void putPreemptedThread(int pri, int thread);

// Remove the first server in the thread run queue.
void removeThread();

// Adjust ready server's priority to the value of the
// released handler
void adjustReadyServer();

// Evaluate the overall number of server threads invoked
void eval();

// Yields the CPU, adjusting the system status
void yield();
```

Listing 3.1: Global Declarations and Functions in the AEH Automata Models

jRate AEH Model

In this section the extended UPPAAL automata model for the AEH implementation in jRate is presented. The original automata model was first introduced in [34]. Here it has been extended, primarily to reduce the complexity of the model and to be compatible with the extended automata model for Java RTS AEH presented in the following section. Listing 3.1 shows the global declarations for variables and channels, and prominently used functions in the jRate automata model. The same set of the global declarations and functions is also used for the Java RTS AEH automata model with two additional synchronisation channels, an extra integer variable, and one more function. The channels and variables are explained along with their usage as they appear. The model of the jRate AEH implementation has two automata, `ServerThread` and `Scheduler` each of which discussed in turn as follows:

`ServerThread` The template, depicted in Figure 3.4, is the automaton of a server thread and represents its main behaviour (i.e. states realised by locations). The template has one formal parameter, a type `threads` which is defined on line 7 in Listing 3.1. The type `threads` is an integer variable with the range [1, `MAXFC`] inclusive. It is passed to the template at the instantiation and a number of instances of the template, defined by `MAXFC`, is created. If `MAXFC` is set to 6, 6 server threads will be created with a unique id set to 1 to 6 for each instance: `ServerThread(1)`, `ServerThread(2)`, ..., `ServerThread(MAXFC)`. The ids are then used with synchronisation channels to identify a specific server thread to communicate with by the `Scheduler` automaton in the jRate AEH automata model. It has four locations: `Waiting`, `Ready`, `Running`, and `Preempted`. Locations in a UPPAAL automata model are connected by edges (or transitions). Edges are annotated with selections, guards, synchronisations and updates, explained as follows:

- Selections: Selections non-deterministically bind a given identifier to a value in a given range. The other three labels of an edge are within the scope of this binding.
- Guards: An edge is enabled in a state if and only if the guard evaluates to true.
- Synchronisation: Processes can synchronise over channels. Edges labeled with complementary actions over a common channel synchronisation.
- Updates: When executed, the update expression of the edge is evaluated. The side effect of this expression changes the state of the system.

The initial location is `Waiting` and a server thread can be one of the four locations (states) at any given time as follows:

1. `Waiting` refers to the state where it is created but has not been made ready yet (i.e. server threads in the thread pool waiting to be invoked is in this state).

2. `Ready` refers to the state where it may be selected to have its state changed to `Running`. In other words, it has been invoked (or notified) by the occurrence of handlers' being released but is still in the thread run queue, waiting to get the CPU.

3. `Running` refers to the state where it is current running on the CPU. When the transition takes place from `Running`, it may proceed to one of two states, `Waiting` or `Preempted`, based on the current status of the AEH system.

4. `Preempted` refers to the state where it is preempted by a higher priority server thread and it may proceed to `Running` when it becomes eligible for execution again (i.e. has the highest priority).

There is an integer array, `st_stat[threads]`, used extensively in the automaton to indicate the current status of server threads. The length of the array is the number of server threads preallocated in the pool (See line 20 in Listing 3.1). The integer variable `MAXFC` is primarily used to limit the number of handlers that may be released for the automata models to have a verifiable bound and is also used to specify the number of server threads to be created for the pool. Therefore each server thread has one array component with its id as an index, representing the current state of the server thread (Recall that the type `threads` is an integer variable in the range [1, `MAXFC`] and therefore there will be `MAXFC` components of the array such that `st_stat[1]`, `st_stat[2]`, ..., `st_stat[MAXFC]`). The values that can be assigned to the array are four constant integer values: `WAITING`, `READY`, `RUNNING`, and `PREEMPTED` (See lines 10-13 in Listing 3.1). There are also four synchronisation channel arrays with the same number of components as that of server threads using the type `threads`: `notify[threads]`, `run[threads]`, `yieldCPU[threads]`, and `preempted[threads]`. With the channel arrays, an automaton that synchronises over the channel can do so with the right automaton among ones that use the same channel. Each server thread uses a component of the channel arrays with its id as an index. Therefore `notify[1]!` synchronises with `notify[1]?` where the number 1 being the id of a server thread. The template starts from the initial location `Waiting`. Every location in the template waits for (or receives) a synchronisation using the label *channelName*`[tid]?` where `tid` represents the id of a server thread. The meaning of the channels are self-explanatory

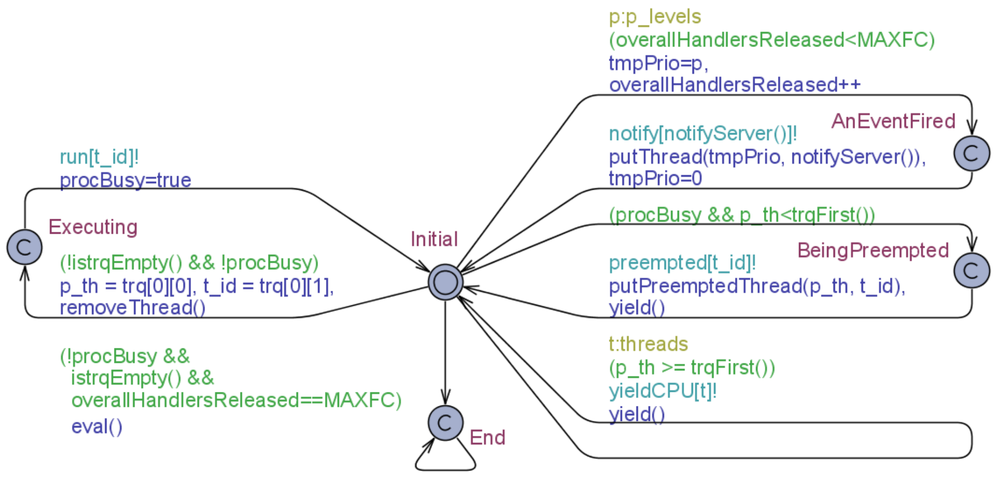

p:p_levels
(overallHandlersReleased<MAXFC)
tmpPrio=p,
overallHandlersReleased++
AnEventFired
notify[notifyServer()]!
putThread(tmpPrio, notifyServer()),
tmpPrio=0
run[t_id]!
procBusy=true
Executing
Initial
(procBusy && p_th<trqFirst())
BeingPreempted
preempted[t_id]!
putPreemptedThread(p_th, t_id),
yield()
(!istrqEmpty() && !procBusy)
p_th = trq[0][0], t_id = trq[0][1],
removeThread()
(!procBusy &&
istrqEmpty() &&
overallHandlersReleased==MAXFC)
eval()
t:threads
(p_th >= trqFirst())
yieldCPU[t]!
yield()
End
C
C
C
C

Figure 3.5: `Scheduler` Automaton for jRate AEH

and they are triggered accordingly by the `Scheduler` automaton. Each transition from one location to another sets `st_stat[tid]` to the appropriate value. In the jRate AEH automata model, various numbers of server threads are created and put into the thread pool when the system is instantiated. They are, in turn, notified to be the leader server thread. An integer array, `invoked[threads]` defined on line 18 in Listing 3.1, is used to keep track of the number of server threads invoked from the pool as increased by 1 whenever they take a transition from `Waiting` to `Ready`. The information will give us the least upper bound and the least lower bound of server threads used to handle a given number of event handlers when verifying the run-time properties (See Section 3.3). When simulated and verified in UPPAAL, 3 to 6 `ServerThreads` are created and put in the thread pool. Note that more than 6 simultaneous `ServerThreads` for the automata models cause state-space explosion which is a common problem in UPPAAL [4]. The verification was performed on a machine with 2048 MB of main memory.

`Scheduler` This is the automaton which represents the base scheduler of the RTSJ (i.e. `PriorityScheduler`) depicted in Figure 3.5. To simplify the overall model, `Scheduler` maintains other architectural modelling components such as the thread pool, thread run queue (`trq[MAXFC][2]`), and pending handler queue (`phq[MAXFC]`) for the Java RTS automata model, which otherwise would have to be modelled in separate automata. This also applies to the Java RTS AEH automata model presented in the next section. `Scheduler` in the jRate AEH automata model maintains two internal data structures: the thread run queue and the thread pool. The thread run queue is a two-dimensional integer array for storing and dispatching server threads in a priority-ordered manner. Each component of the array therefore has two sub-components that are used for each waiting server thread in the thread run queue to store its information; the first column is used to indicate the priority and the second column represents the id of a server thread. The thread pool data structure simulates normal thread pool operations. It has one user-visible operation, `notifyServer()`, that is used comprehensively by this automaton. The operation looks for a waiting server thread in the pool by searching incrementally and invoke the first found one. The template has four outgoing transitions from the initial location, `Initial`. Each transition is explained below:

1. The transition to the location `AnEventFired`: This transition represents that a handler is released as the result of an event firing. The transition has a selection annotated (`p:p_levels`), which non-deterministically binds `p` to an integer in the range set by `p_levels` (See line 5 in Listing 3.1). `p_level` is a type like `threads` and is set to an integer value in the range [1, `PRIORITY_LEVELS`], representing the priority of a handler. Therefore `p` can be an integer in the range 1 to `PRIORITY_LEVELS`, inclusive. When the transition takes place, a priority level is non-deterministically bound to `p`, representing the released handler's priority. The transition updates an integer variable `tmpPrio` to the value of the released handler's priority level `p` for the later use and increments `overallHandlersReleased` by one, which keeps track of the overall number of released handlers. The variable is used to give the AEH automata models a verifiable bound (i.e. if `overallHandlersRelea sed` becomes equal to or greater than `MAXFC` the transition is disabled, binding the overall number of handlers within a certain value). The location `AnEventFired` is committed (marked with a `c`). Committed locations in UPPAAL are the most restrictive in the sense that time may not progress and interleavings with normal locations are not allowed in them. Therefore one of outgoing transitions from a committed location must take place without delay or interleavings with other locations (urgent and normal). They are frequently used in the models to establish happens-before relationships and to reduce the state space for the verification. The outgoing transition from the location `AnEventFired` then makes the invoked server thread (leader) ready by synchronising with the corresponding `ServerThread` automaton over the channel `notify[notifyServer()]!` and puts the server thread in the thread run queue.

The above two transitions collectively and conceptually represents that the environment randomly fires events. When an event is fired, the occurrence of the event with the associated handler's priority level is informed to `Scheduler`. Then it notifies a server thread in the pool and the invoked server is made ready with the released handler. Therefore it is assumed that server threads put in the thread run queue have their respective handlers bound to them.

2. The transition to the location `Executing`: This transition is guarded by the constraints `(!istrqEmpty() && !procBusy)`, which corresponds to the state where the thread run queue is not empty and the CPU is idle. Therefore the transition is enabled whenever there is a pending server thread in the thread run queue and the CPU is not occupied. The thread run queue is always kept in a priority-ordered manner. `Scheduler` therefore always executes the highest priority thread first. On the transition two integer variables, `p_th` and `t_id` defined on line 22 and 23 in Listing 3.1, that respectively represent the priority and the id of the current running server thread, are updated accordingly by taking the first waiting thread out from the thread run queue. The variables are used throughout the execution of the server thread to identify the current running server thread and the priority of it when requested. The next transition from the location `Executing` to the initial location immediately takes place, synchronising with the corresponding server thread automaton over the channel `run[t_id]!`. As a result, the server thread automaton moves its location from `Ready` to `Running`. It also updates the status of the system accordingly (i.e. `procBusy`, defined on line 21 in Listing 3.1, that represents the boolean CPU state occupied or not is set to true).

3. The transition with the channel `yieldCPU[t]!`: The transition is guarded by the constraints (`p_th` $\geq$ `trqFirst()`), which corresponds to the state where the current running server thread has the highest priority. This means that the current running server thread finishes the execution of the bound handler and therefore it yields the CPU. The transition also updates all the relevant information using the operation `yield()` such that `procBusy` is set to false, `p_th` and `t_id` are set to 0, and `CS` is incremented by one. The integer variable `CS` is to keep track of the number of context-switches incurred for the verification purpose. The transition conceptually puts the yielding server thread in the pool for the next use, synchronising with the current running server thread over the channel `yieldCPU[t]!`.

4. The transition to the location `BeingPreempted`: As soon as the guard (`procBusy &&` `p_th`$<$`trqFirst()`) on the transition to the location `BeingPreempted` is satisfied,

which corresponds to the state where there is a higher priority server thread ready in the thread run queue while a lower priority one occupies the CPU, the transition is enabled. Then the outgoing transition from the location `BeingPreempted` is immediately performed (as committed), synchronising with the corresponding automaton of the server thread over the channel `preempted[t_id]!` and updating the status of the AEH system accordingly by invoking the operation `yield()`. As the result of the synchronisation, the corresponding server thread automaton moves its location from `Running` to `Preempted`. Note that, the `putPreemptedThread()` method, which operates on the thread run queue, puts the server thread being preempted at the front of all pending server threads with the same priority in the thread run queue.

5. The transition to the location `End`: This transition is designed in `Scheduler` in order to provide well-defined life-time of the AEH automata model. When `overallHandlersReleased` reaches the predefined value `MAXFC`, the CPU is idle, and there are no server threads pending in the thread run queue, then the transition will take place, allowing the system to halt safely (i.e. event firings are no longer allowed and the only outgoing transition from the location `End` is self-looping). The operation `eval()` is performed in the transition, which evaluates the overall number of server threads invoked and the overall number of server threads required. Therefore the resulting values from the operation are used to derive the overall number of server threads required to execute handlers under a particular handler releasing scenario (See Section 3.3). The operation is as follows:

```
void eval() {
   int i=1;
   while(i <= MAXFC) {
      if(invoked[i] >= 1) {
         // Ignores multiple invocations
         overallSTsRequired++;
      }
      // Counts all invocations
      overallSTsInvoked = overallSTsInvoked + invoked[i];
      i++;
   }
}
```

Java RTS AEH Model

In this section the extended UPPAAL automata model for the AEH implementation in Java RTS is presented. The original automata model was introduced in [34]; here it has been extended, primarily to show more detailed behaviour when a server changes its

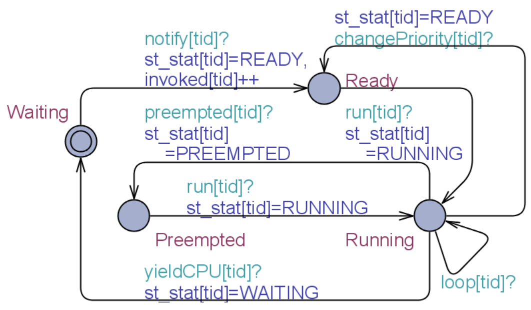


Figure 3.6: `ServerThread` Automaton for Java RTS AEH

priority. Readers are referred [34] for the original AEH automata model for Java RTS. Here we discuss only different features designed for the Java RTS AEH automata model from the jRate one presented in the previous section. Most components used in the Java RTS AEH model such as the environment, server thread, scheduler and thread pool are essentially the same as those in the jRate AEH model, with minor differences. However a pending handler queue is used in this AEH automata model as released handlers are first stored in the pending handler queue before the binding to a server thread. There is only one queue for the entire AEH model and therefore it is priority-ordered. Although the actual Java RTS AEH implementation has one queue per priority level, the behaviour of the single-queue approach in the Java RTS AEH model is identical with the multiple-queue approach in the actual implementation as long as the queue is kept in a priority-ordered manner. This is again to reduce the state space when verifying the model.

`Server Thread` The automaton model, depicted in Figure 3.6, represents the main behaviour of server threads in Java RTS AEH. There are two additional transitions from the location `Running`. One goes back to the location `Ready`, synchronising with `Scheduler` over the channel `changePriority[tid]?`. The other one is a self-looping transition that synchronises on the channel `loop[tid]?`. Note that server threads in the AEH automata model wait for (or receive) a synchronisation, not send (or fire) one. The transitions are added to reflect the Java RTS AEH implementation that it allows server threads to execute more than one handler and to change their priorities dynamically to that of the handler to which they are currently bound. The two additional transitions work as follows:

1. The self looping transition from the location `Running`: This is required in the Java RTS AEH automata model as it allows server threads to execute more than one handler as long as the condition specified in `Scheduler` with the transition `Initial` to `Looping` is satisfied. The channel in this automaton waits for the synchronisation.

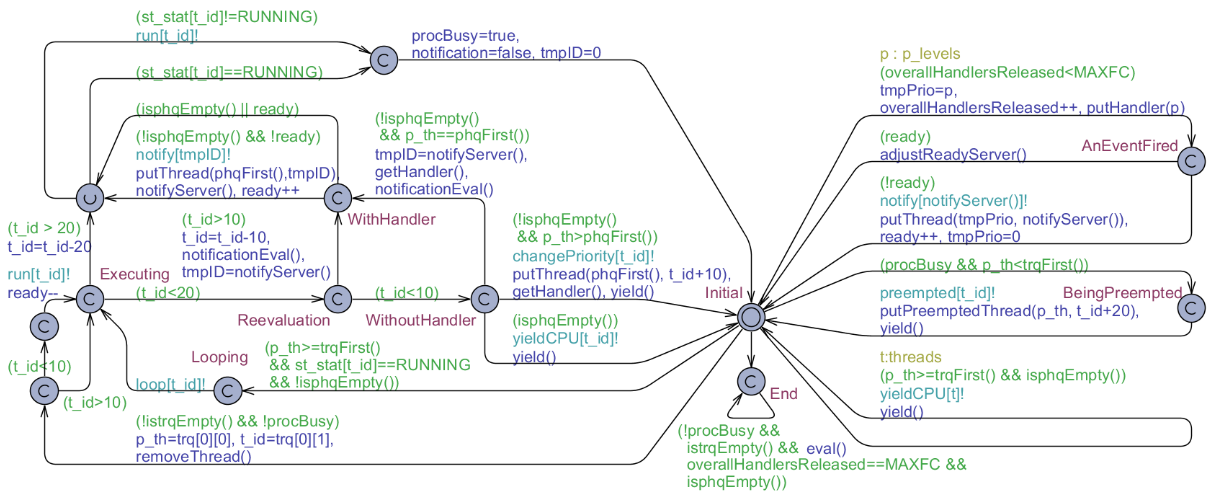


Figure 3.7: `Scheduler` Automaton for Java RTS

The transition conceptually illustrates that the current running server thread finishes the execution of its currently bound handler and continues the execution with the first handler in the pending handler queue.

2. The transition from `Running` to `Ready`: When a server thread removes the first handler from the pending handler queue, it changes its priority to that of the removed handler if the priority of the server thread is higher than that of the removed handler. When a thread changes its priority it is added to the tail of the thread run queue within the same priority range. This transition models this behaviour and makes the current running server thread ready from running by adding the server to the tail of the thread run queue for its new priority. The effect of the priority change of server threads are discussed in detail later in this section.

`Scheduler` The automaton in Figure 3.7 depicts the base scheduler of the RTSJ (i.e. `PriorityScheduler`) in Java RTS. `Scheduler` has six outgoing transitions from the initial location, synchronising with server thread automata over various channels. `Scheduler` in the Java RTS AEH model more complex (i.e. more transitions) than the counterpart in the jRate AEH model. This is mainly because the AEH implementation in Java RTS allows server threads to execute more than one handler once it is invoked until all pending handlers in the queue are consumed. This behaviour requires more information to be available at run-time for the AEH system to work correctly. The followings explain each outgoing transition in the automaton from the initial location `Initial` and they differ (more or less) from the transitions presented in the jRate `Scheduler` automaton:

1. The transition to the location `AnEventFired`: The transition has an additional update, `putHandler(p)`, which enqueues the released handler in the pending handler queue, reaching to the location `AnEventFired`. Now there are two possible outgoing transitions from `AnEventFired` to the initial location based on the existence of a ready server thread. If there is the ready server thread, the `adjustReadyServer()` operation is performed, in which the priority of the ready server is adjusted to that of the released handler if the priority of the ready server thread is lower than that of the released handler. Otherwise (i.e. there is no ready server), `Scheduler` notifies a new ready server via `notify[notifyServer()]!`, and updates the system status accordingly (i.e. `ready++`). The integer variable, `ready`, is used to keep track of the number of ready server threads in the system (and also to check the existence of a ready server) and is expected to always be equals to or lower than 1.

2. The transition to the location `BeingPreempted`: The transition is identical to that in the jRate AEH `Scheduler` automaton. However, in the outgoing transition from the location `BeingPreempted` the method `putPreemptedThread()` puts the server thread being preempted at the head of the queue at its priority and it adds 20 to its id, `t_id`, to indicate merely that this particular server thread was preempted. This is done to distinguish between waiting server threads in the thread run queue based on their status (e.g. was it preempted or not?).

3. The transition with the location `Looping`: The transition is guarded by the constraints `(p_th≥trqFirst() && st_stat[t_id]==RUNNING && !isphqEmpty())` that corresponds to the state where the priority of the current running server still has the highest priority and there is one or more pending handlers in the queue. It then proceeds to the location `Executing`, synchronising with the corresponding server thread over the channel `loop[t_id]!`. As a result of the transition the corresponding server thread automaton takes the self-looping transition, representing the condition under which it is allowed to continue execution with another handler at the head of the pending handler queue.

4. The transition to the location `Executing`: The transition is identical to that in the jRate AEH `Scheduler` automaton, updating the two integer variables, `p_th` (representing the current running server's priority) and `t_id` (representing the server's

id), and removing the first server thread from the queue. The transition leads to the anonymous committed location where two possible outgoing transitions are defined based on the current status of the removed server thread. A server thread conceptually has a property of telling us its state, whether it yielded the CPU as the result of changing its priority (i.e. 10 is added to its id), was preempted (i.e. 20 is added to its id), or is a new ready server. If the server taken from the queue yielded due to the priority change or was preempted, then `Scheduler` is able to execute it from where it left off. If the variable is less than 10 (meaning that the server is a newly notified), the server thread is made running and decreases `ready` by one, synchronising the corresponding server thread over the channel `run[t_id]!`. Otherwise it proceeds to `Executing` without the synchronisation as it should be further evaluated. From the location `Executing`, one transition out of two outgoing ones is taken, again based on the evaluation of `t_id`. It now tells us whether the server yielded the CPU as a result of priority change or was preempted. If it was preempted (i.e. `t_id`>20), it goes up and executes from where it stopped the execution of its handler. Otherwise it proceeds to the next location, `Reevaluation`, where there are two possible outgoing transitions. Here the `Scheduler` makes a choice, again based on reevaluating `t_id`. If it is greater than 10, which means that the server already has a handler to execute bound to the server, it proceeds to the location `WithHandler`, where it determines whether to notify another server in the pool based on the current state of the AEH system. If `t_id` is less than 10, it proceeds to the location `WithoutHandler` which has three outgoing transitions. If the handler queue is empty (i.e. `isphqEmpty()`), the server yields the CPU and goes back to the pool. If the queue is not empty and the priority of the first handler in the queue is lower than that of the server, the server takes the first handler out and changes its priority to that of the handler. Hence it is placed back at the tail of the handler queue via the channel synchronisation, `changePriority[t_id]!` with the corresponding server thread [5][4]. In the

[4]The RTSJ states that for a `schedulable` object whose active priority is changed as a result of explicitly setting its base priority, this `schedulable` object is added to the tail of the queue for its new priority level. This is somewhat contrary to what the RTSJ tries to address for the base scheduler (i.e. it calls for the FIFO priority scheduler to not force periodic context switching between tasks of the same priority and therefore the base scheduler in the RTSJ is meant to minimise the number of context-switches between same priority threads). However the semantics and requirements for priority changes defined in the RTSJ unintentionally make the context-switching more frequent when `schedulable` objects change their priorities. See Section 3.3.4 The Multiple-Server Switching Phenomenon for a more detailed explanation about this behaviour.

transition it adds 10 to its id, `t_id`, to show this particular server thread has a handler to execute so that `Scheduler` is able to execute the server thread from where it left off later on. Otherwise it proceeds to the location `WithHandler` where the server notifies another server in the pool only when there are one or more pending handlers in the queue and no ready server. Otherwise it continues without the notification. Now the server thread is finally running, going back to the initial location.

3.3 Formal Analysis of the Models

In this section we specify several properties for the models defined in Section 3.2. The AEH automata models are simulated and verified using a simplified version of CTL (Computation Tree Logic) in UPPAAL. The query language in CTL consists of path formula and state formula. State formula describes individual states, whereas path formula quantify over paths or traces of the model. In path formula, A and E mean every path and some path, and [ ] and <> represents all states and some state, respectively. For example A[ ] φ asks whether the given formula φ is always true in all reachable states, whereas E[ ] φ says that there should exist a maximal path such that φ is always true. E<> φ can be expressed as there exists a maximal path such that φ is true in some state. Furthermore, $\rightarrow$ represents the *leads to* or *response* property, written $\phi \rightarrow \varphi$ which is read as whenever ϕ is satisfied, then eventually φ will be satisfied. There are generally 3 forms of properties when verifying a formal model: reachability, safety and liveness. Reachability properties are the simplest form of properties. They ask whether a given state formula, φ, possibly can be satisfied by any reachable state. In UPPAAL, they are expressed using the syntax E<> φ. Safety properties are on the form that something bad will never happen. In UPPAAL, they are expressed using the syntax A[ ] and E[ ]. Liveness properties are of the form that something will eventually happen. In Uppaal these properties are written as A<> and $\phi \rightarrow \varphi$. Readers are referred to [4] for a more detailed explanation about the query language for UPPAAL. The set of constructed properties for the model can be classified into three criteria and the following shows some of the queries we used to verify the properties. Note that simple reachability properties verified successfully against the model are not listed here.

1. Model consistency: The properties of this category guarantee that the models comply with the RTSJ.

2. Model safety: The properties of this category guarantee that the models are free

from unwanted behaviour and give us their respective worst case scenarios.

3. Run-time behaviour: The properties of this category show us one possible path of the exhaustive dynamic behaviour of the AEH models during run-time. One path is presented for each AEH, which is obtained using the UPPAAL simulator. This allows us to observe their respective run-time behaviour such as the mapping, notification rule and relinquishment rule as the path progresses. For example, the path presented for Java RTS AEH produce a MSSP in order for us to observe its cause at run-time.

3.3.1 Model Consistency

- Property 1: A server with a higher priority will always execute in preference to a server with a lower priority when both are eligible for execution.

  procBusy==true && Scheduler.Executing → p_th>=Scheduler.trqFirst()

- Property 2. The currently executing server thread will immediately be preempted by a higher priority ready thread.
  a. procBusy==true && Scheduler.BeingPreempted → p_th<Scheduler.trqFirst()
  b. E<> procBusy && Scheduler.AnEventFired && p_th<Scheduler.trqFirst()
  c. E<> procBusy && Scheduler.Executing && p_th<Scheduler.trqFirst()
  d. E<> procBusy && Scheduler.End && p_th<Scheduler.trqFirst()
  In order to prove this property, all the above four queries (a - d) should be considered as a whole. This is because the tempting query to prove this property

  p_th<Scheduler.trqFirst() → procBusy==true && Scheduler.BeingPreempted

  will not hold (i.e. when `p_th<Scheduler.trqFirst()` is satisfied `Scheduler` does not necessarily have to move to the `BeingPreempted` location due to non-determinism). Therefore we first check the query a which asks that whenever the `BeingPreempted` location is reached, the priority of the first server in the thread run queue is always higher than the priority of the current running thread. The second query b checks if there is a path where the priority of the first thread in the thread run queue is higher than that of the current running server and the `Scheduler` automaton is in the `AnEventFired` location. Similarly the third and fourth queries, c and d, are constructed. a and b hold true and c and d are not satisfied. The results can be interpreted such that when the priority of the first server in the queue is higher than that of the current running thread, only two actions, event firing and preemption,

are possible. Therefore from the results of the queries, it can be concluded that the current running thread will be immediately preempted by a higher priority ready server, as an event firing should be considered asynchronous.

- Property 3. The release of a handler may occur regardless of the current status of the AEH models at any time.

  1. After the release of and before the execution of the previously released handler:

     E<> Scheduler.AnEventFired && phqlen > 1

  2. Before the completion of the previously released handler that is currently being executed:

     E<> p_th==i && Scheduler.AnEventFired && tmpPrio==j

  3. After the completion of the previous handler:

     E<> Scheduler.AnEventFired && invoked[1]==1 && !procBusy && phqlen==1 && trqlen==0

- Property 4. A preempted server will always be placed at the head of the run queue for its priority level.

  ServerThread(n).Running && p_th==i && Scheduler.BeingPreempted $\rightarrow$ trq[1][1]-20==n && Scheduler.trqFirst() > i

  The variable n and i represent the id and the priority of the current running server, respectively. Therefore the above query is read as if the current running server n with the priority i is preempted, the priority of the thread in front of n is always higher than that of n, which is i.

- Property 5. The server thread that changes its priority will be added to the tail of the thread run queue within the range of its priority.

  ServerThread(n).Running && p_th==2 && Scheduler.WithoutHandler && p_th > Scheduler.phqFirst() && !Scheduler.isphqEmpty() $\rightarrow$ trq[trqlen-1][0]==1 && trq[trqlen-1][1]-10==n

  The variable n represents the id of the current running server. This query is designed to check for the situations where the server thread changes its priority to the lowest. Other priority changes, such as 4 to 3, cannot readily be checked as it is not possible to determine the position of the server thread added to the tail for its new priority. However the query partially guarantees this property, along with Property 6 below.

- Property 6. The pending handler queue (phq) and the thread run queue (trq) is always performed in a priority-ordered manner.

For trq: A[ ] trq[0][0]>=trq[1][0] && trq[1][0]>=trq[2][0] && ... && trq[n-1][0]>=trq[n][0]

For phq: A[ ] phq[0]>=phq[1] && phq[1]>=phq[2] && ... && phq[n-1]>=phq[n]

- Property 7: When an event occurs its attached handler is released for execution.

tmpPrio==3 → Scheduler.phqFirst() == 3

3.3.2 Model Safety

- Property 1. The AEH models are always guaranteed free from a deadlock.

A[ ] not deadlock

- Property 2. There is always zero or one server thread running on the CPU at any point in time.

A[ ] ServerThread(1).Running + ServerThread(2).Running + ... + ServerThread(n).Running <= 1

- Property 3 for only the Java RTS AEH model: There is only one or none ready server at any point in time.

A[ ] ready ≤ 1

- Property 4. The least upper and lower bounds of server threads required as the number of released handers increases.

E<> Scheduler.End && overallSTsRequired==i && overallSTsInvoked==i

To observe how many server threads are required for executing a certain number of handlers we vary the two constant integer variables, `PRIORITY_LEVELS` and `MAXFC`, for each verification. They vary from 3 to 6 and this effectively constraints the AEH environment that releases handlers. If `PRIORITY_LEVELS` and `MAXFC` are set to

	PRIORITY_LEVELS	MAXFC	Max STs (i)	Min STs (i)
jRate AEH	3	2	2	2
		3	3	3
		4	4	4
		5	5	5
	4	3	3	3
		4	4	4
		5	5	5
		6	6	6
	5	4	4	4
		5	5	5
		6	6	6
	6	5	5	5
		6	6	6
Java RTS AEH	3	2	2	2
		3	3	2
		4	4	2
		5	4	2
	4	3	3	2
		4	4	2
		5	5	2
		6	5	2
	5	4	4	2
		5	5	2
		6	6	2
	6	5	5	2
		6	6	2

Table 3.1: The Number of Server Threads Required as The Number of Priority Levels or Concurrently Active Handlers Increases

3 and 3 respectively, it restricts the environment such that the maximum number of handlers that can be released is 3 and their priority levels can be in the range [1,3] inclusive. With those numbers it generates 3 × 3 × 3 (hence 27) different but possible scenarios of handlers' releasing as follows (the higher the number the higher the priority):

```
1–1–1  2–1–1  3–1–1
1–1–2  2–1–2  3–1–2
1–1–3  2–1–3  3–1–3
1–2–1  2–2–1  3–2–1
1–2–2  2–2–2  3–2–2
1–2–3  2–2–3  3–2–3
1–3–1  2–3–1  3–3–1
1–3–2  2–3–2  3–3–2
1–3–3  2–3–3  3–3–3
```

If the timing property is also included in the combinations, it becomes $(3{\times}1){\times}(3{\times}3)$ ${\times}(3{\times}3) = 243$ possible scenarios. This is because the first released handler conceptually represents that it is released after the previous handler completes but the second and the third handler may be released before the previous handler starts executing, in the middle of the previous handler being executed, or after the previous handler completes. If those numbers for `PRIORITY_LEVELS` and `MAXFC` increase, scenarios that the verifier in UPPAAL generates become considerably larger. Among those possible event firing scenarios the verifier therefore gives us the worst and the best event firing combination in terms of the number of server threads required. The variable i indicates the least upper and lower bounds of server threads required. If the properties is satisfied with i set to 3, this means that there is a event firing scenario with `PRIORITY_LEVELS` and `MAXFC` set to a certain value, where it requires 3 server threads and they are explicitly invoked. For example if `overallSTsRequried` is 1 and `overallSTsInvoked` is 3 means that a server thread was invoked three times and it executed all the released handlers. However this scenario depicts that handlers are released in a discrete manner where a handler is released after the completion of the previous handler. The variable i and two variables, `overallSTsRequired` and `overallSTsInvoked`, are therefore introduced to eliminate the scenarios that involve discrete (not simultaneous) event firings. In such event firing combinations only one server thread is needed as there is always one released handler to execute.

The results from the two AEH automata models are shown in Table 3.1. Recall that the blocked state is absent in the automata models in order for them to be given a well-defined cycle of execution. The jRate AEH automata model requires the same number of server threads as the number of simultaneously released handlers regardless of the scenario of event firings. As the number of handlers increases, the number of server threads required is also increasing. The least upper and lower bounds of server threads required are the same for the model and it shows the run-time behaviour of the dynamic 1:1 mapping in general. Therefore the number of server threads required in the jRate AEH automata model entirely depends on the number of released handlers, not on the handlers' blocking or priority levels in the system. On the other hand, the Java RTS AEH automata model requires $p + 1$ server threads in the worst case, where p represents the number of priority levels in the system. If the number of handlers in the system is equal to or less than p, the least upper bound of server threads required is p or less in the Java RTS AEH model. However the least lower bound number is 2 regardless of the number of released handlers. The best cases are observed only when released handlers have the same priority level as this does not require server threads to change their priorities, in which the MSSP (See Section 3.3.4) cannot be seen and all other scenarios require more than 2 server threads due to the MSSP. Note that, if all handlers block, it is easy to see that the Java RTS AEH implementation will require the same number of server threads as that of the released handlers.

- Property 5. The least upper and lower bounds of context-switches incurred to serve a certain number of handlers.

$$\text{E<> Scheduler.End \&\& CS==}j$$

The operation `yield()` increments the variable `CS` by one and it is called whenever a context switch between server threads is made. The variable j is replaced with a natural number when verifying this property. The results from three models are shown in Table 3.2. To observe how many context-switches are incurred for a certain number of released handlers we vary the two variables `PRIORITY_LEVELS` and `MAXFC` as done for verifying the previous property. The least upper bound of the context-switches are all the same for the models and increases as the number of released handlers increases. This is because the context-switches are incurred when the running server thread yields or is preempted. The worst case number of preemption will occur when the low priority thread is released first and other events are released in a priority-ordered manner before the event completes. Therefore the worst case number of context-switches is:

	PRIORITY_LEVELS	MAXFC	Max CS (j)	Min CS (j)
jRate AEH	3	2	3	3
		3	5	5
		4	7	7
		5	9	9
	4	3	5	5
		4	7	7
		5	9	9
		6	11	11
	5	4	7	9
		5	9	9
		6	11	11
	6	5	9	9
		6	11	11
Java RTS AEH	3	2	3	2
		3	5	2
		4	7	2
		5	7	2
	4	3	5	2
		4	7	2
		5	9	2
		6	9	2
	5	4	7	2
		5	9	2
		6	11	2
	6	5	9	2
		6	11	2

Table 3.2: The Number of Context Switches Incurred as The Number of Priority Levels or Concurrently Active Handlers Increases

$$\sum_{j\in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil + n$$

where $hp(i)$ is the set of higher priority threads than i, R_i is the worst case response time of the thread i, T_j is the period of the thread j [6] and n is the number of active server threads. Here it is assumed that the first task that gets the CPU does not incur a context-switch as the CPU was empty[5]. If there are 6 handlers released, the worst case context-switches is 11. This applies to all the models. Again the least lower bound of the context-switches for the Java RTS automata model, which is 2 regardless of the number of released handlers, is observed only when released handlers have the same priority level as this does not require server threads to change their priorities, in which the MSSP cannot be seen. All other scenarios require more than 2 context-switches.

3.3.3 Run-time Behaviour

The run-time behaviour of the AEH automata models is obtained by the UPPAAL simulator. The simulator in UPPAAL is a validation tool that enables examination of the possible dynamic executions of a system during early design (or modelling) stages. In this sense, it provides an inexpensive mean of fault detection prior to verification by the model-checker. The simulator is also used to visualise executions (i.e. symbolic traces) generated by the verifier. The Message Sequence Chart (MSC) panel is used to provide the visulisation and displays an MSC view of the randomly generated trace by the simulator (guided trace by the user is also possible by manually selecting one of enabled transitions). Figures 3.8 to 3.11 shows the MSC snapshots of randomly generated trace for the AEH automata models presented in Section 3.2. In the MSC view there is a vertical line for each process, and a horizontal line for each synchronisation point. The process name of each vertical line is indicated in the upper part of the MSC panel. The node names shown on the vertical lines indicate the control location of the automaton. If the *random* option is selected in the simulator, it starts a random simulation where the simulator proceed automatically by randomly selecting enabled transitions.

A run-time behaviour of the jRate AEH automata model

Figure 3.8 shows the MSC for a randomly generated run-time scenario of the jRate AEH automata model by the UPPAAL simulator. The two variables `PRIORITY_LEVELS` and

[5]The exact schedulability analysis that also models context-switch times is referred to Section 4.5.

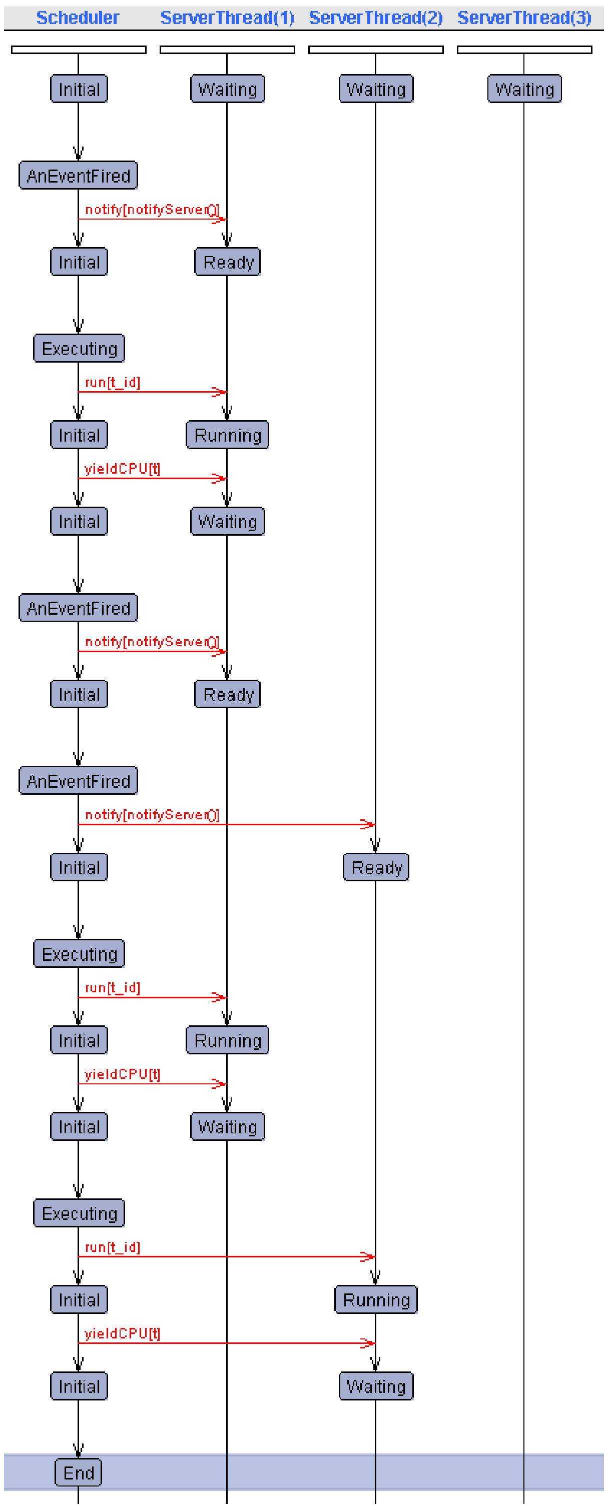


Figure 3.8: A Random Simulation for jRate AEH automata model

MAXFC is both set to 3 and therefore 3 handlers are released with three different priority levels. There are three server threads waiting in the pool. First Scheduler detects a handler releasing and therefore notifies ServerThread(1). It makes the server thread running and the server thread yields the CPU after the completion of the handler, going back to the waiting state. Scheduler detects another handler releasing and notifies ServerThread(1) again as it is in the Waiting state. The third (and therefore the last) handler is released and Scheduler notifies ServerThread(2). Now there are two server threads in the thread run queue. ServerThread(1) is made running first as its priority is higher than the ServerThread(2). After the completion of ServerThread(1), ServerThread(2) is made running and executes its bound handler. Lastly it yields the CPU. In the location End, the variables overallSTsInvoked and overallSTsRequired are set to 3 and 2, respectively. This means that for this combination of 3 handlers being released, 2 server threads are required with 3 invocations. As discussed earlier these resulting values indicate that there were discrete handlers releasing as the two values are not equal. This is true because the second handler is released after the completion of the first handler and therefore ServerThread(1) is invoked twice.

A run-time behaviour of the Java RTS AEH automata model

Figures 3.9 to 3.11 show the MSC for a randomly generated run-time scenario of the Java RTS AEH automata model by the UPPAAL simulator. They collectively depict a single scenario where a latter chart immediately follows its former one. The variables PRIORITY_LEVELS and MAXFC are also set to 3, having 3 handlers to be released. The first released handler has the priority 1 and Scheduler notifies ServerThread(1) upon the release of the handler. The second handler is then released with the priority 1 and there is the ready server thread, ServerThread(1). Therefore the ready server still keeps its priority as the priority of the released handler is the same as that of the ready server. Recall that according to the late binding model that the Java RTS uses, when a handler is released, no further actions are required if there is a ready server and the priority of the server is equal to that of the released handler. The third handler is released with the priority 2 and therefore the AEH subsystem increases the ready server's priority to 2. Now there is the ready server with the priority 2 and 3 pending handlers in the queue. Then ServerThread(1) is made running and takes the first handler in the pending handler queue. Before it starts executing the handler it notifies ServerThread(2) as there are two pending handlers in the queue. ServerThread(2) is made ready at the priority of the first handler at the time, which is 1 and ServerThread(1) starts executing the handler. After

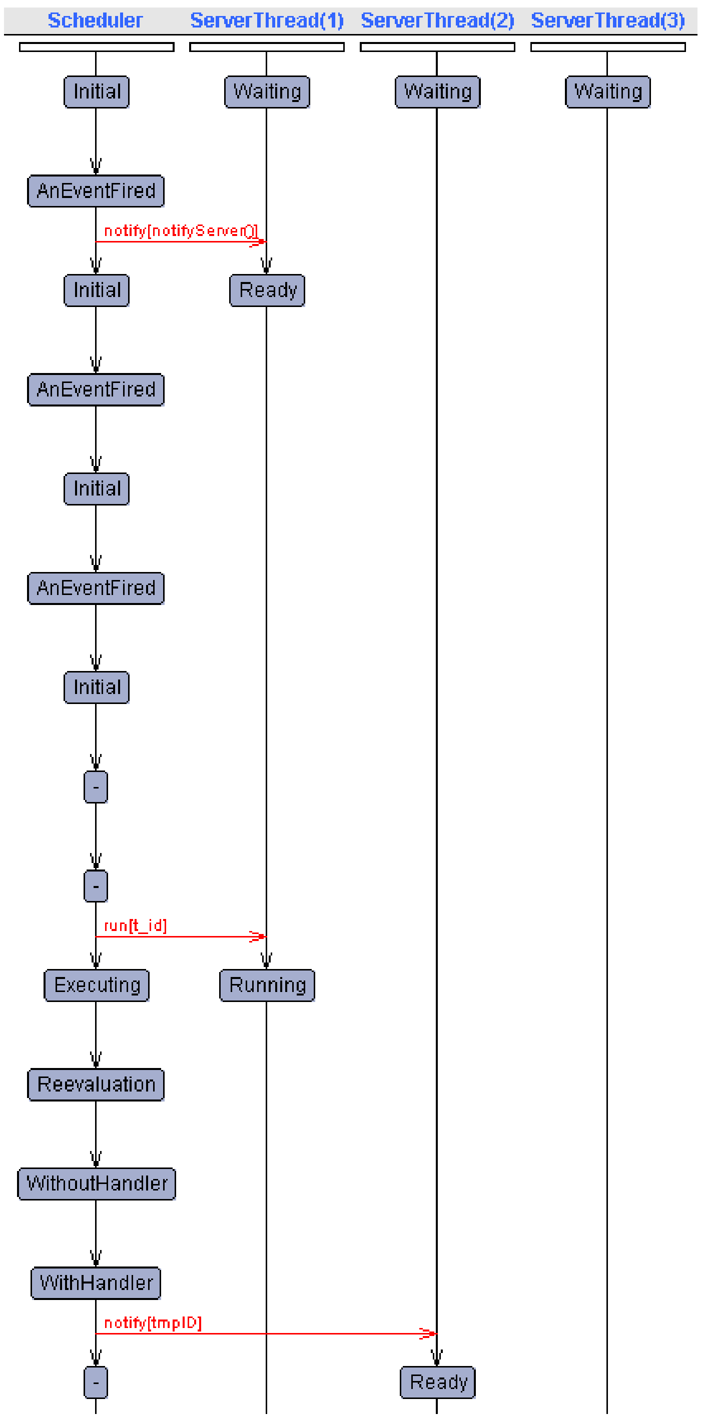


Figure 3.9: A Random Simulation for Java RTS AEH automata model I

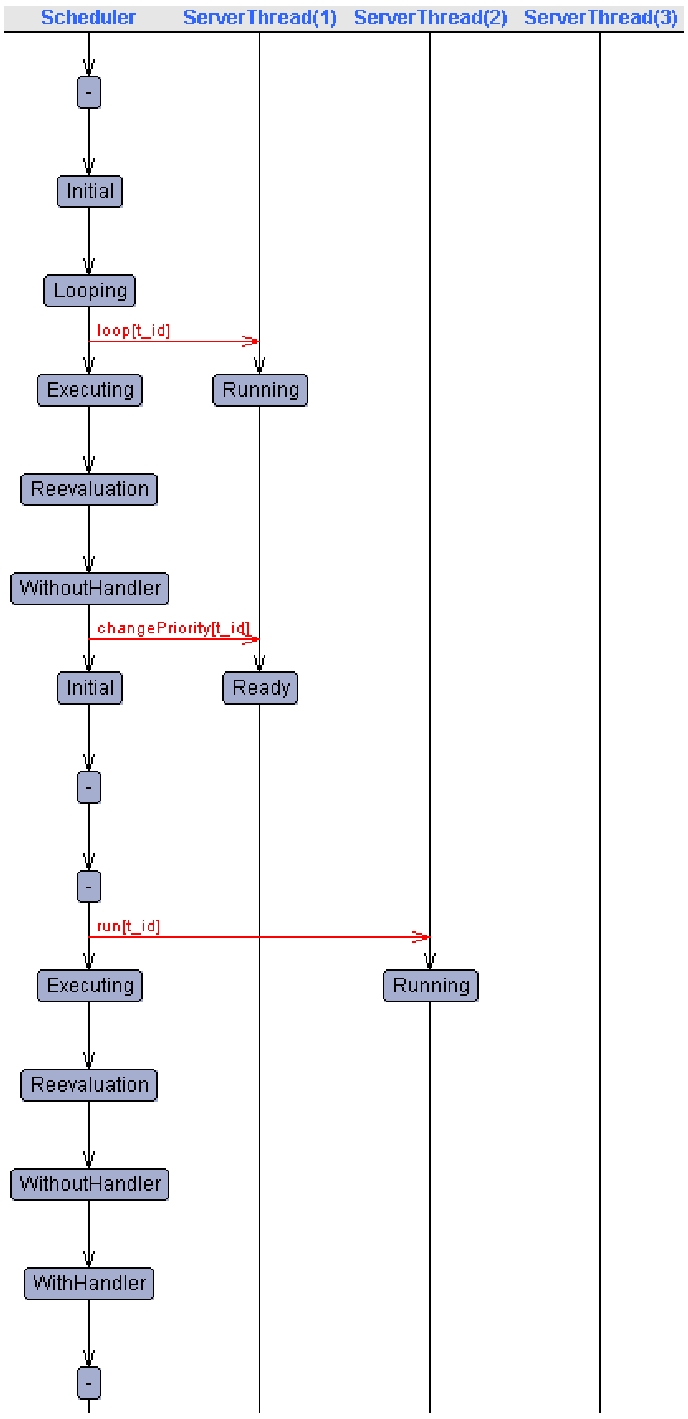


Figure 3.10: A Random Simulation for Java RTS AEH automata model II

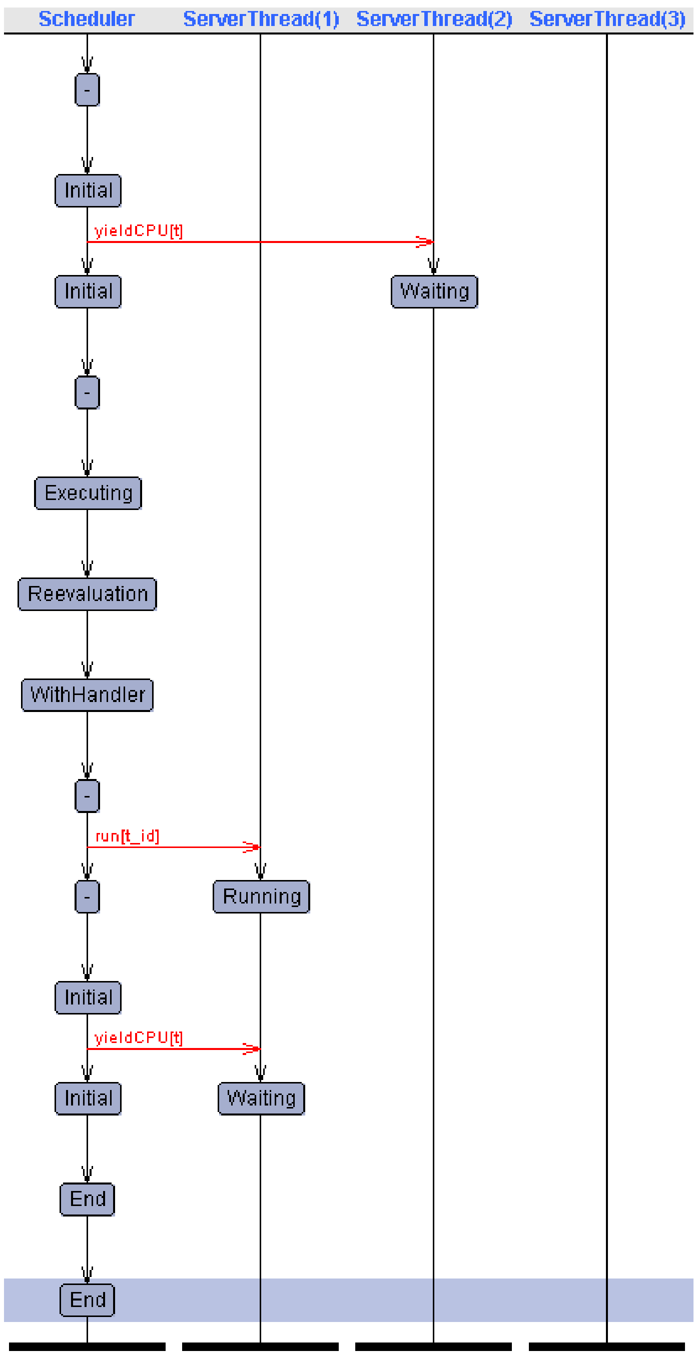


Figure 3.11: A Random Simulation for Java RTS AEH autamata model III

the completion `ServerThread(1)` loops and takes the first handler from the queue. As a binding process, the current server adjust its priority to that of the taken-out handler which is 1. Now there is a ready server in the thread run queue with the priority 1 and hence the current server is put to the tail of the thread run queue due to the priority change[6]. The ready server therefore gets the CPU and takes the last remaining handler in the queue. As its priority is the same as the last handler in the queue, it starts to execute the handler without a priority change. After the execution `ServerThread(2)` yields the CPU as there are no pending handlers in the queue. `ServerThread(1)` now again gets the CPU and executes its bound handler. Therefore the overall server threads required is 2 to execute 3 released handlers with 2 unnecessary context-switches for this particular handler releasing sequence.

3.3.4 The Multiple-Server Switching Phenomenon

The AEH automata model for Java RTS has demonstrated that it tries to reduce the number of server threads required by allowing server threads to execute multiple handlers under certain circumstances. As shown in the previous section, there are however occasions the AEH model in Java RTS constantly causes unnecessary context-switches, which we call the *Multiple-Server Switching Phenomenon (MSSP)*. The phenomenon occurs inevitably whenever the currently executing server changes its priority to the value of the first handler in the pending handler queue to execute the handler. Having one or more pending handlers in the queue, there is a ready server with the priority of the first handler in the queue according to [50]. Therefore the ready server will immediately preempt the currently executing server when the current server changes its priority to that of the first handler in the queue as the ready server will be at the same priority level. This is because the RTSJ defines that a schedulable object that changes its active priority is added to the tail of the run queue for its new priority level.

The UPPAAL model checking tool also provides run-time simulation of automata models, not only the best and worst case behaviour by exhaustively searching every path of automata models. Using the simulator in the tool, we are able to find the scenarios that involve the MSSP. This subsection explains one of the simplest scenarios that causes the phenomenon. Assume that the thread pool is created with a sufficient number of server threads and the AEH subsystem has just started. Also assume that we have 3 pending

[6]This is called the multiple-server switching phenomenon, which is explained in Section 3.3.4 in more detail.

handlers with different priority levels in the queue, namely `H3`, `H2` and `H1` (The last numeric character of the name of handlers indicates the priority of them and the higher the number the higher the priority). Hence there is a ready server, `S1`, at the priority 3. When the ready server gets the CPU, it takes the first handler, `H3`, from the queue and checks the queue. There are pending handlers, `H2` and `H1`. So it notifies one of the server threads, `S2`, in the pool and `S2` is made ready with the priority 2. Then `S1` starts to execute `H3`. After the completion of the handler, `S1` checks the pending handler queue and finds out that the queue is not empty. Thus it takes the first handler, `H2`, from the queue and adjusts its priority to 2. Now the running server, `S1`, is added to the tail of the thread run queue for its new priority. Hence `S2` gets the CPU and polls the first handler in the queue, which is `H1`. As soon as `S2` sets its priority to 1, it will be preempted by `S1` of the priority 2. Then `S1` will continue to execute the handler, `H2`. Here we have two additional context-switches in order for `S1` to execute two handlers, `H3` and `H2`. This phenomenon is constantly recurring when the currently executing server lowers its priority to the value of the first handler in the queue and there is the ready server, which is waiting at the priority of the first handler in the queue. Furthermore, the AEH subsystem now does not have a ready server as the server threads, `S1` and `S2`, now have the handlers to execute. Therefore if a handler is released at these instants, another server will be notified and be made ready. The MSSP is also seen in the best case scenarios, where 2 server threads are required regardless of the number of released handlers. For example when there are three pending handlers, each of which has a unique priority, in the pending handler queue, 2 server threads are required with 2 unnecessary context-switches. As discussed here, the AEH model in Java RTS tries to reduce the number of server threads required on average by allowing server threads to execute more than one handlers in turn. Regardless of the number of the released handlers in the system it requires 2 server threads unless higher priority handlers are released or handlers block (or self-suspend) while executing in the best case as shown in Table 3.1. However it seems that the implementation causes the multiple-server switching phenomenon that produces extra context-switches on average and demands another ready server thread as a result. The phenomenon occurs due to the following reasons:

1. For the implementation-defined threads, the RTSJ underspecified their behaviour at run-time, and

2. The operation that sets a thread's priority typically causes the thread that changed its priority to be put at the tail of the thread run queue for its new priority.

This phenomenon must be addressed to fully utilise the dynamic 1:N mapping model for an efficient and predictable AEH implementation of the RTSJ.

3.4 Summary

In this chapter we first investigated asynchronous event handling algorithms used in various RTSJ implementations. The followings summarise their respective approaches:

- RI - Dynamic 1:1 mapping with run-time server creation and destruction overhead
- OVM - Static 1:1 mapping
- Jamaica - Static 1:N mapping
- jRate - Dynamic 1:1 mapping using the Leader/Followers design pattern
- Java RTS - Dynamic 1:N mapping using a thread pool

As discussed in Chapter 2, the asynchrony used to distinguish the distributed communication models is based on synchronisation, time, and space decoupling. Recall that synchronisation decoupling means that producers are not blocked while producing events, and consumers can be asynchronously notified of the occurrence of an event while performing some concurrent activity. Space decoupling indicates that the interacting parties do not know each other. Time decoupling shows that the interaction between producers and consumers do not need to be actively participating the interaction at the same time. Decoupling of the above three criteria removes all explicit dependencies between the interacting participants and enables the resulting communication application to be well adapted to distributed environments that are asynchronous. The only AEH implementation which offers both synchronisation and space decoupling is provided in Java RTS by using a thread pool, although time decoupling is not achieved in the implementation. However, note that, applying the notion of time decoupling to real-time systems is not appropriate as asynchronous events and their handlers must complete within deadlines. Java RTS AEH is also the only implementation that tries to reduce the overall number of server threads required on average without causing priority-inversion.

To evaluate asynchronous event handling of the two RTSJ implementations, jRate and Java RTS, UPPAAL has been extensively used. The two AEH automata models for each implementation have been designed, simulated and verified. They have been shown to be consistent with the RTSJ and the properties including reachability, safety and liveness have been verified successfully. The verifier in UPPAAL has generated some measurements for resource usage and overheads using the AEH automata models. The verification results show that the jRate AEH model uses the same number of server threads as the number of simultaneously released handlers, regardless of the sequence of event firings. The Java RTS AEH model, on the other hand, require a smaller number of server threads on average although the same number of server threads are required in the worst case. The results from the simulation and the verification demonstrates that the AEH model used in Java RTS is relatively efficient by allowing server threads to execute multiple handlers under certain event-firing sequences. However, there are occasions when the Java RTS AEH model constantly causes unnecessary server-switching, multiple-server switching phenomenon (MSSP), incurring more context-switches on average. The MSSP can be classified as a concurrency-related issues (such as priority inversion) and takes place whenever the current running server changes its priority to that of the handler, which it will next execute, due to queue replacement policy in that most real-time operating systems 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 Java RTS AEH implementation in order for the current running server to reflect the priority of the handler that it currently executes. In order for the AEH implementation to be more efficient and hence more scalable it is necessary to effectively deal with the MSSP. Note that the MSSP is not specific to Java RTS AEH. It can be found in any system where more than one server thread is concurrently active and their priorities are changed dynamically. The next chapter presents and discusses a logical approach to derive the least upper bounds of server threads for blocking and non-blocking handlers. Based on this, two new AEH models are developed to use the least upper bounds of server threads in the worst case while avoiding the MSSP.

Chapter 4

Efficient AEH for the RTSJ

Asynchronous event handlers in the RTSJ are primarily designed to execute a few lines of code and then exit while threads are intended to execute for a long time and expected to wait for resource, sleep, change its priority, generally taking full advantage of the available support [16, 37]. This explains the foremost driving goal of AEH in the RTSJ, which is to realise a lightweight concurrency mechanism that is capable of support hundreds (if not thousands) of handlers. Consequently handlers should not incur the same overheads as real-time threads [5, 16, 69]. Therefore handlers are not expected to wait, block, sleep, or change their scheduling parameters; notwithstanding, the RTSJ permits these possibilities, allowing handlers to call self-suspending code (e.g. `wait()`, `sleep()`, blocking I/O operations, etc.). As a consequence all the AEH algorithms used in the RTSJ implementations discussed in Chapter 3 are designed based on the assumption that handlers might *block* (or *self-suspend*) in the course of execution, except the Jamaica's AEH implementation. This effectively prevents handlers from being lightweight as it is necessary to allocate a single server to each released handler to avoid priority inversion for meaningful schedulability analysis. In many applications, most handlers will not self-suspend and therefore can be handled in a more efficient way, enabling them to be more lightweight (i.e. using fewer servers on average). Fewer servers result in smaller data structures in the real-time virtual machine, less stack space for server threads, fewer context-switches between server threads etc. The AEH implementation challenge in meeting these goals is made virtually impossible because the RTSJ allows a handler to block (or self-suspend) during its execution. Hence, a server thread executing a handler may not be able to complete its execution without suspending. Once suspended, it is unable to execute any other handler. Therefore if each handler does not have a real-time thread for its execution, priority inversion will be incurred. For example, if the server dedicated to a priority level blocks, the remaining pending handlers (if any) at the priority level will also block until the server unblocks. In the mean time, a server associated with a lower priority level will run, causing priority inversion. It should be noted, however, that the potential delay incurred when accessing a synchronised statement or method is not considered as a suspension. This is because priority inheritance will cause another handler already bound to a server to execute – the one that has the required lock[1].

In Chapter 3, the AEH models used in some popular RTSJ implementations were classified based on their run-time behaviour (i.e. how they map handlers to server threads at run-time). With a 1:1 mapping model both dynamic and static, it is not possible to realise the main motivation of non-bound asynchronous event handlers in the RTSJ. The static 1:N mapping model does reduce the number of servers to execute a greater number of handlers. For example, Jamaica allocates a dedicated server per priority level. Therefore it requires the same number of servers as that of the priority levels, not that of handlers. However the model is subject to priority inversion if handlers block (or self-suspends). Hence when using Jamaica, all blocking handlers must be designated as bound handlers. Of course, there is no run-time check to catch the case where a handler does block, and hence unbounded priority inversion can still silently occur. The AEH implementation in Java RTS is classified as a dynamic 1:N mapping as they allow a server to execute as many handlers as possible without causing any undesirable behaviour such as unbounded priority inversion. However, as identified in the previous chapter, there are occasions when the Java RTS AEH model constantly causes unnecessary context-switching between server threads, the MSSP.

In this chapter, two new AEH models are developed for blocking and non-blocking handlers to achieve the lightweightness requirement of AEH in the RTSJ. They use fewer server threads on average and their least upper and lower bounds of server threads are also derived. It is also shown that the new AEH models outperform other AEH models used in the current RTSJ implementations. In this chapter, we first examine the dynamic 1:N mapping to determine the worst case handler releasing sequence that requires the least upper bound of server threads. Based on the results of this, two new AEH models for

[1]On a multiprocessor system this may not be the case. The lock that the current running thread on one processor tries to acquire may already be possessed by a thread on another processor. This may result in the former processor being idle until the thread holding the lock releases it.

the RTSJ are proposed, namely *blocking AEH* and *non-blocking AEH*. The handling of blocking and non-blocking handlers is kept separate in order to take advantage of non-blocking handlers. With blocking handlers it is generally required to have an additional server waiting at the priority of the first handler in the queue to take over the CPU if the currently running server blocks as a precautionary measure. This is not the case with *non-blocking AEH*. The common underlying idea for both models is to enable servers to execute multiple handlers as long as they do not cause any concurrency-related side effects such as unbounded priority inversion, run-time server creation and destruction overhead, and the MSSP. It is also discussed how the two new AEH models effectively remedy the MSSP and the other concurrency related side-effects.

4.1 Critical Sequence for Dynamic 1:N Mapping Model

Mapping models are based on the following facts derived from AEH in the RTSJ such that:

- Handlers require servers to execute.
- Once a handler starts to execute, the server cannot execute other handlers until the current handler finishes. In other words, a server can execute only one handler at any time.

The above statements collectively define the ground upon which the mapping between servers and handlers can be established. On top of this, the mapping model must also meet the following additional requirements.

- **Requirement 1**: It must preserve the priority-based execution of handlers in order to carry out some form of priority-based schedulability analysis,
- **Requirement 2**: It must prevent any concurrency-related issue such as priority inversion, unbounded blocking, and the MSSP without a breach of Requirement 1, and
- **Requirement 3**: It should use the least number of servers for a given number of handlers without a breach of Requirement 2.

Therefore, the most efficient approach to the dynamic 1:N mapping is to use the existing servers to execute as many pending handlers as possible whenever it can preserve the priority-based execution of handlers and prevent any concurrency-related issue. Requirement 1 can be achieved by dynamically adjusting the priority of the current running server to that of the handler to be executed. For Requirement 2, it is imperative to define a set of rules for the condition under which the invocation and the relinquishment of servers should take place. To prevent priority inversion and unbounded blocking it is necessary to invoke additional servers. However as more than one active server at the same time is the source for the MSSP, any additional servers must be invoked wisely and the number of active servers should be kept to a minimum at all times. Requirement 3 can be realised by invoking a server only when absolutely necessary in conjunction with Requirement 2.

Based on the requirements of the general servers and handlers framework, the least upper and lower bounds of servers required for a given set of handlers can be derived by considering all possible handler releasing scenarios. For example, assuming handlers do not self-suspend and there are two handlers (`H1` and `H2`) to be released, there are 9 possible releasing scenarios as follows:

- **Scenario 1**: `H2` has a lower priority than that of `H1`
- **Scenario 2**: `H2` has the same priority as that of `H1`
- **Scenario 3**: `H2` has a higher priority than that of `H1`

Each scenario has 3 possible release sequences as follows:

- **A**: `H2` is released at the same time as `H1` (i.e. the critical instance).
- **B**: `H2` is released in a discrete manner such that `H2` is released at or after the completion of `H1`.
- **C**: `H2` is released after `H1` starts executing and before `H1` completes.

For example Scenario 2.A represents the handler releasing sequence such that two handlers with the same priority are released at the same time. Note that there is another possible handler releasing sequence, Scenario D, where `H2` is released before `H1` is released. This scenario, however, is precluded from the set of handler releasing scenarios as it is already accounted for (For example Scenario 1.D is equivalent with Scenario 3.B or 3.C). Depending on the scenario and the number of handlers, the number of servers required may or may not increase. Figure 4.1 depicts all the handler releasing sequences as the number of released handlers increase. At each table, the first column represents the scenario and the second column shows the number of servers required on the corresponding scenario. Initially only one server is required with one released handler `H1`. When `H2` is released, 8 scenarios do not require an additional server. As for example Scenario 1.C, that

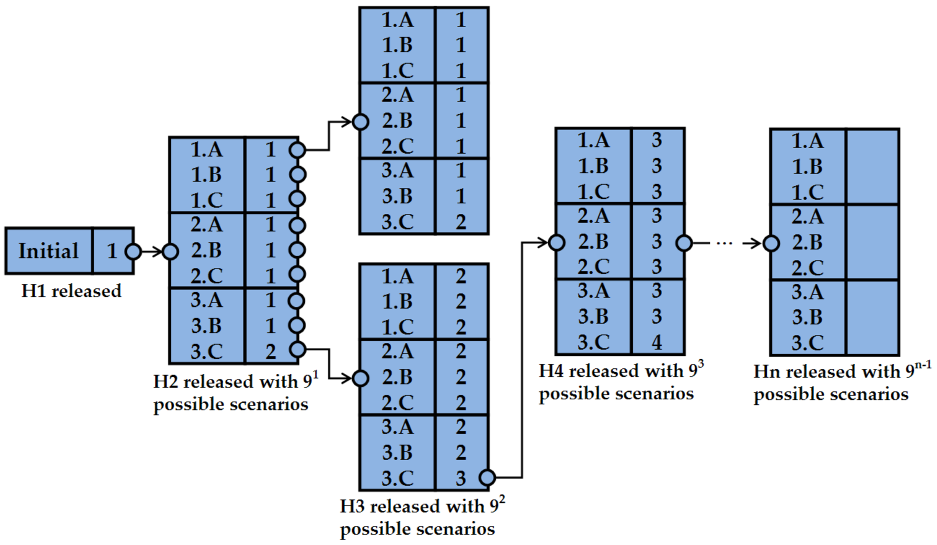

Initial
1
H1 released
1.A
1.B
1.C
2.A
2.B
2.C
3.A
3.B
3.C
H2 released with 9^1 possible scenarios
H3 released with 9^2 possible scenarios
H4 released with 9^3 possible scenarios
Hn released with 9^{n-1} possible scenarios

Figure 4.1: All Possible Scenarios in an Iterative Manner

is `H2` with a lower priority is released after `H1` with a higher priority starts executing and before `H1` completes, can be handled with one server while meeting all the requirements.

On the other hand, Scenario 3.C which reads as `H2` with a higher priority is released after `H1` starts executing and before `H1` completes, requires an additional server to prevent priority inversion; Without the additional server priority, inversion will occur in Scenario 3.C as a higher priority handler `H2` will be blocked until `H1` finishes. To prevent this priority inversion, it is necessary to have another server ready to execute `H2` as soon as it is released to meet Requirements 1 and 2 and therefore Scenario 3.C requires 2 servers.

Now assume that there is another handler `H3` to be released. Each release scenario has the same 9 nested scenarios possible, resulting in 81 possible scenarios as shown in Figure 4.1. Among all the possible scenarios, only the sequence, from Scenario 3.C to nested Scenario 3.C, requires an additional server with `H3` releasing. All the other scenarios require 1 or 2 servers, regardless of the number of handlers. This is because the scenario that requires 3 servers is such that `H1` is released first, `H2` is released before `H1` completes, and `H3` is released after `H2` starts executing and before `H2` completes, requiring 3 servers to cater for 2 preemptions as shown in Figure 4.2. This set of 9 scenarios can be iteratively applied until all handlers are released. By enumerating all possible scenarios for a given number of handlers, it is shown that the number of servers increases as the number of priority levels increases. This is because only Scenario 3.C, which involves a higher priority

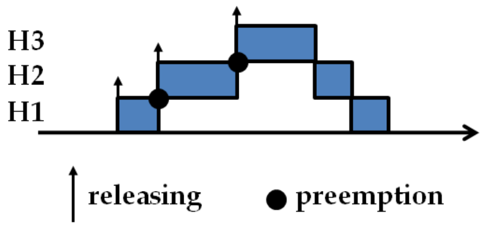


Figure 4.2: 2 Preemptions for 3 Priority Levels

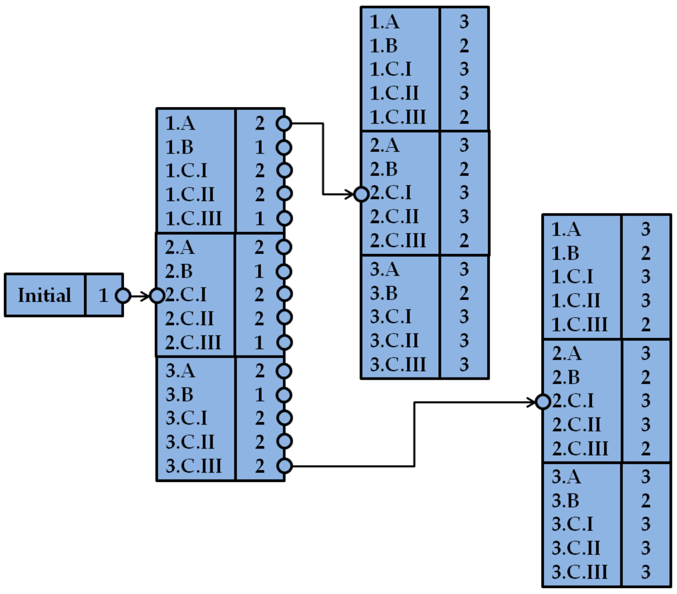


Figure 4.3: Numbers of Servers for Blocking Handlers

handler, requires an additional server. We call such a sequence (sequential happenings of Scenario 3.C) that demands the least upper bound of servers the *critical sequence* for the dynamic 1:N mapping model. Deductively, the following defines the condition under which a critical sequence of handler releases $\mathbf{H} = \{H_i, H_{i-1}, ..., H_2, H_1\}$ occurs:

> In a fixed priority system where handlers do not self-suspend, a critical sequence of the handler releases occurs when handlers are released in a priority-ascending manner in turn and one is released in the middle of a previously released handler being executed, that is, r_i is in the range $[r_{i-1}, r_{i-1} + C_{i-1}]$ exclusively, for every handler in $\mathbf{H}$. (Where r_i and e_i denote the release and the execution time of H_i and the greater the value i, the higher the priority.)

Based on the critical sequence, therefore, the least upper bound of servers required is equal to the number of priority levels in the system for non-blocking handlers. Consequently the value N in the dynamic 1:N mapping is the number of released handlers per priority level for non-blocking handlers in the worst case. For the best case, it requires only one server. Of course, this assumes FIFO scheduling within the same priority level. If there is a requirement for round-robin or EDF scheduling within priority, more servers will be required.

If the restriction placed on the self-suspension is lifted, the scenario C should be nested as follows to cater for self-suspension:

- **I**: `H2` is released before `H1` self-suspends.
- **II**: `H2` is released during the self-suspension of `H1`.
- **III**: `H2` is released after `H1` resumes the execution after the self-suspension.

As a result, there are 15 exhaustive scenarios for the two handlers releasing as shown in Figure 4.3 and the number of servers required no longer depends on the number of priority levels. The number of servers required is now dependent on the number of handlers. All scenarios will require to assign a single server for each handler that self-suspends to prevent unbounded blocking, except when handlers are released in a discrete manner, Scenario 1.B, 2.B, and 3.B and when `H2` has a lower or equal priority and is released after `H1` resumes the execution after the self-suspension, Scenario 1.C.III and 2.C.III. Deductively, the following defines the condition under which a critical sequence of a set of blocking handlers $\mathbf{H}$ occurs:

> In a fixed priority system where every handler self-suspends, a critical sequence of handler releases for handler set $\mathbf{H}$ occurs when a handler is released after or at the release time of a previous handler, and before or during the self-suspension of the previous handler, that is, r_i is in the range $[r_j,\ r_j + ss_j]$ exclusively, for every handler in $\mathbf{H}$. (Where ss_j denotes the finishing time of H_j's self-suspension.)

Therefore, if all handlers are potentially self-suspending, The value N in the dynamic 1:N mapping becomes 1 in the worst case as each handler requires a single server, regardless of the number of priority levels in the system. If blocking and non-blocking handlers are mixed, and the blocking handlers can be identified at run-time by the RT-JVM, it is possible to assign a single server per blocking handler and schedule non-blocking handlers using either a static or dynamic 1:N mapping. Based on the critical sequence of the 1:N mapping model for blocking and non-blocking handlers, therefore, the least upper bound of servers required ω_s of the system in the worst case is given by

$$\text{if } \pi_s + \beta_s < \psi_s, \text{ then } \omega_s = \pi_s + \beta_s \tag{4.1}$$

$$\text{if } \pi_s + \beta_s \geq \psi_s, \text{ then } \omega_s = \psi_s \tag{4.2}$$

where π_s denotes the number of priority levels, β_s represents the number of blocking handlers, and ψ_s indicates the total number of handlers in the system.

4.2 Blocking AEH

The AEH implementation in Java RTS assumes that handlers may block. Under this assumption, it tries to reduce the number of server threads required on average by using the late binding technique and having an additional server thread as a precautionary measure for the current running thread's blocking. Note that, in Section 4.1, we classified scenarios into three categories with respect to self-suspension, **I**, **II**, and **III**. If `H2` is released after the self-suspension of `H1` and `H2` has an equal or lower priority than that of `H1`, the current running server can be used to execute `H2` after the completion of `H1`. In practice, however, it is not possible to detect or project when a handler will self-suspend. Therefore the Java RTS AEH and blocking AEH implementations take a precautionary measure by having a ready server thread, if there is one or more pending handlers for the case when the current running server blocks.

While the goal of Java RTS AEH is laudable, its implementation reduces the efficacy of the approach due to the MSSP. The blocking AEH implementation [35] is proposed primarily to prevent the MSSP while using the least possible number of server threads. Therefore the blocking AEH implementation employs a set of more restrictive rules than the Java RTS AEH implementation, regarding the run-time behaviour of servers, in order to satisfy Requirement 1, 2, and 3 shown in the previous section. The set of rules is as follows:

1. A server that lowers its priority to that of the handler, is added to the head of the thread run queue for its new priority.
2. The current running server thread is allowed to execute another handler after the completion of the current handler if and only if it is the only server in the AEH subsystem except the ready server. Otherwise it yields the CPU.

3. After the completion of the current handler, the running server lowers the priority of the ready server to that of the second pending handler in the queue if and only if it is allowed to continue the execution with the first handler in the queue (i.e. the condition specified in the second rule holds).

This set of rules for blocking AEH collectively prevents the MSSP found in the Java RTS AEH implementation. The first rule is to enable the current running server not to be preempted due to priority change by the ready server thread waiting at the priority of the first handler to which the current server thread will change its priority. The second rule is to prevent the MSSP between the current running server and preempted servers. Consider the case where the current running server thread has preempted the previous running server and therefore there is a preempted server in the thread run queue at a lower priority than that of the current server. If a handler with a lower priority than that of the preempted thread is released at this instance, another server will be made ready as there is no ready server. If the current running server is allowed to execute further, there will be a MSSP entailed as the preempted server thread becomes to have a higher priority than the current running server when it changes its priority to that of the first pending handler in the queue, which is lower than that of the highest pending server thread. In order to prevent this, the current server thread yields the CPU if there are more than one server thread, except the ready server, in the AEH subsystem. The last rule is also designed to prevent the MSSP between the current running server thread and the ready server thread in conjunction with the first rule. The first rule only prevents the MSSP between servers with the same priority. Therefore the current running server that has just finished the execution of a handler lowers the ready server's priority to that of the second handler in the pending handler queue. As a result the current running server can continue the execution as now the priority of the ready server will be equal to or lower than the current running server. This allows the current running server to continue the execution while keeping the ready server still ready even after the current server's priority change. Based on this set of rules, the AEH UPPAAL automata model for Java RTS has been extended, which is presented in Section 4.2.1. The extended AEH automata model is called blocking AEH. The actual implementation of the blocking AEH model is presented in Section 4.2.2.

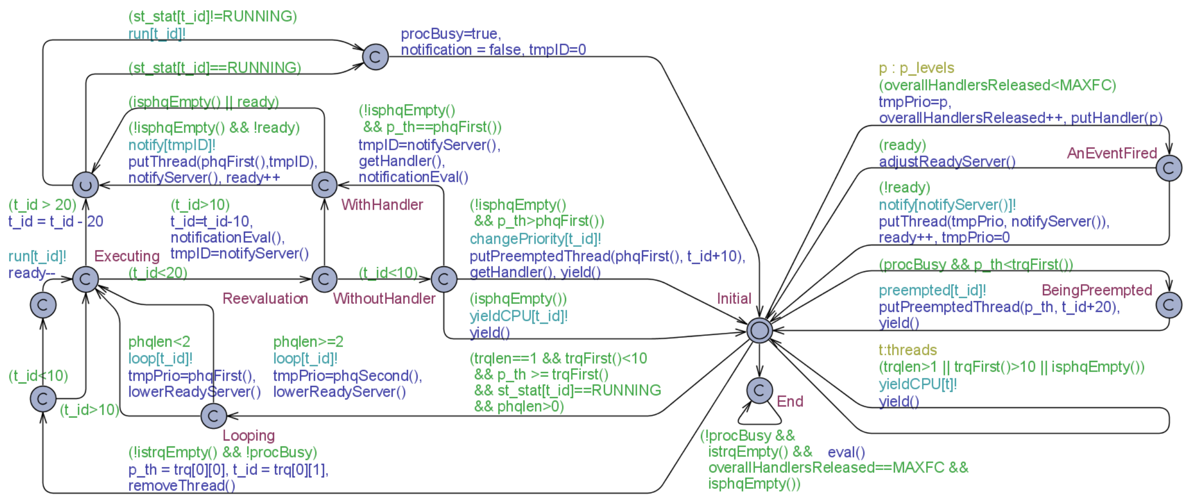


Figure 4.4: `Scheduler` Automaton for Blocking AEH model

4.2.1 UPPAAL Automata Model for Blocking AEH

Essentially the blocking AEH model extends the Java RTS AEH model presented in Section 3.2. The `ServerThread` automaton used for the blocking AEH model is the same as that used in the Java RTS AEH model. However the `Scheduler` automaton used in the blocking AEH model has been modified to reflect the set of rules for blocking AEH. Therefore only the `Scheduler` automaton is discussed in this section.

The Scheduler Automaton

The automaton in Figure 4.4 represents the base scheduler of the RTSJ (i.e. `PriorityScheduler`) for the blocking AEH model. The automaton also has six outgoing transitions from the initial location, synchronising with the `ServerThread` automata. Some transitions are identical with ones in the `Scheduler` automaton for Java RTS AEH shown in Figure 3.7. Hence we only discuss the different transitions from the previous ones as follows.

1. The transition from the location `WithoutHandler` to `Initial`: The transition refers to the edge triggered when the handler queue is not empty (`!isphqEmpty()`) and the priority of the current server is higher than that of the first handler in the queue (`p_th>phqFirst()`). This means that the current server will change its priority to that of the first handler. When the current server is put in the thread run queue as the result of the priority change, the server is placed at the head of the queue for its new priority by using the `putPreemptedThread()` operation. Although this is

	PRIORITY_LEVELS	MAXFC	Max STs (i)	Min STs (i)
Blocking AEH	3	2	2	2
		3	3	2
		4	4	2
		5	4	2
	4	3	3	2
		4	4	2
		5	5	2
		6	5	2
	5	4	4	2
		5	5	2
		6	6	2
	6	5	5	2
		6	6	2

Table 4.1: Numbers of Server Threads and Blocking AEH

not a preempted thread, the effect of invoking the operation achieves how it should behave when changing its priority. Other operations done in the edge are the same.

2. The transition from the location `Initial` to `Looping`: The current running server is allowed to continue the execution if and only if the current running server is the only server in the AEH subsystem except the ready server (i.e. `trqlen==1 && trqFirst()<10`).

3. The transition from the location `Looping` to `Executing`: If the number of pending handlers is greater than or equal to 2 (`phqlen`>=2), the ready server's priority is set to the priority level of the second handler in the queue.

Formal Analysis of the Blocking AEH model

The blocking AEH model is simulated and verified using the same set of the properties that are applied for the Java RTS AEH model. The Figure 4.1 shows the verification results of the least upper and lower bounds of servers required as the number of released handlers increases. This is identical with the results produced by the Java RTS AEH model. Therefore the blocking model, as shown in Table 4.1, requires $p + 1$ server threads in the worst case, where p represents the number of priority levels in the system. If the number of handlers in the system is equal to or less than p, the least upper bound of server threads required is p in the blocking AEH model. However the least lower bound is 2 regardless of the number of released handlers. Recall that blocking is not allowed in the automata models in order for them to be given a well-defined cycle of execution: of course if blocking is allowed, it is shown earlier that the same number of server threads is required as the number of released handlers. In the Java RTS automata model, the best cases are observed only when released handlers have the same priority level as this does not require server threads to change their priorities, in which the MSSP cannot be seen and all other scenarios require more than 2 server threads due to the MSSP. However in the blocking AEH model the best case also includes the handler releasing sequences where handlers are released in a priority-descending ordered manner. The blocking AEH model is also capable of handling the MSSP which inevitably occurs whenever a server changes its priority. The following subsection explains how the blocking AEH model avoids the MSSP using the same scenario used to demonstrate the phenomenon. The scenario is also based on a simulation path of the model using the UPPAAL simulator.

Avoiding the MSSP

The same handler set used to explain the MSSP is employed to demonstrate how the blocking AEH model avoids the phenomenon while using the least possible number of servers[2]. There are three handlers pending in the queue, `H3`, `H2`, and `H1`, in a priority ordered manner (The higher the number, the higher the priority). Therefore a server is made ready at the priority 3 (The highest priority level at the time). When the ready server, `S1`, gets the CPU, it unregisters itself from the ready server and it takes the first handler, `H3`, out from the queue and tries to change its priority to the value of the removed handler. Here as the priority of the server and the handler is the same, the currently running server continues occupying the CPU. Before starting to execute the handler, the server checks the pending handler queue. As there are two pending handlers, `H2` and `H1`, in the queue, it notifies another server in the pool to be the new ready server. Hence a server, `S2`, wakes up and sets its priority to 2, the priority of the first handler in the queue at the time. After `S1` finishes executing `H3` and before going back to the start of the loop, it lowers the ready server's (S2) priority to that of the second handler in the queue, 1 at the time. As done in the previous round, `S1` takes the first handler, `H2`, and sets its priority to 2. Unlike previous instance, now there is the ready server. Hence `S1` skips the

[2]The source code also contains the symbolic trace of this scenario. (BlockingAEHModel_Avoiding_MSSP.xtr)

calling for another ready server and changes its priority 3 to 2 to reflect the priority of the current handler at hands. Here `S1` is not preempted by the ready server, which is done in the Java RTS AEH model, as `S1` lowered `S2`'s priority to that of the second handler which is 1 before it changes its priority. Consequently, the server is allowed to continue running with `H2`. After the completion of `H2`, `S1` now cannot change the ready server's priority as the second handler in the queue is null. Again it goes back to the start of the loop and takes the first handler, `H1`, from the queue. The server sets its priority to 1. Here `S1`'s priority becomes the same as that of the ready server. However `S1` is put to the head of the thread run queue and therefore it is allowed to continue the execution. If it was put to the tail of the thread run queue for its new priority, it would have been preempted by the ready server, `S2`, which was entailed by the Java RTS AEH model. In the blocking AEH model, however, `S1` continues the execution with the third handler, `H1`, as `S1` is not preempted as it is placed at the head of the queue for its new priority. However the self context-switch introduced here when a thread changes its priority is inevitable. Furthermore the ready server can still be the ready server (if a handler is released at these instants, the AEH subsystem does not have to notify another server). This may also benefit from data locality of the system due to cache affinity [14] by allowing the currently running server to continue the execution. As a result of this, the blocking AEH model prevents the MSSP which introduces additional context-switches. This applies to any monitor execution eligibility control provided by the RTSJ (i.e. `PriorityInheritance` and `PriorityCeilingEmulation`) [5]. As demonstrated using the simple scenario, the blocking AEH model uses two servers to execute three handlers without the MSSP while being able to have a ready server for newly released handlers by using a synchronisation block.

4.2.2 Blocking AEH Implementation

Figure 4.5 shows the interaction between relevant objects in the blocking AEH implementation using a sequence diagram. In the figure there are two separate sectors, `Sector 1` and `Sector 2`, depicting application-specific and implementation-specific part of the model, respectively.

Sector 1

This section depicts the application-specific part of the AEH model. The programmer is responsible for constructing relevant objects as needed. In this sector, the programmer

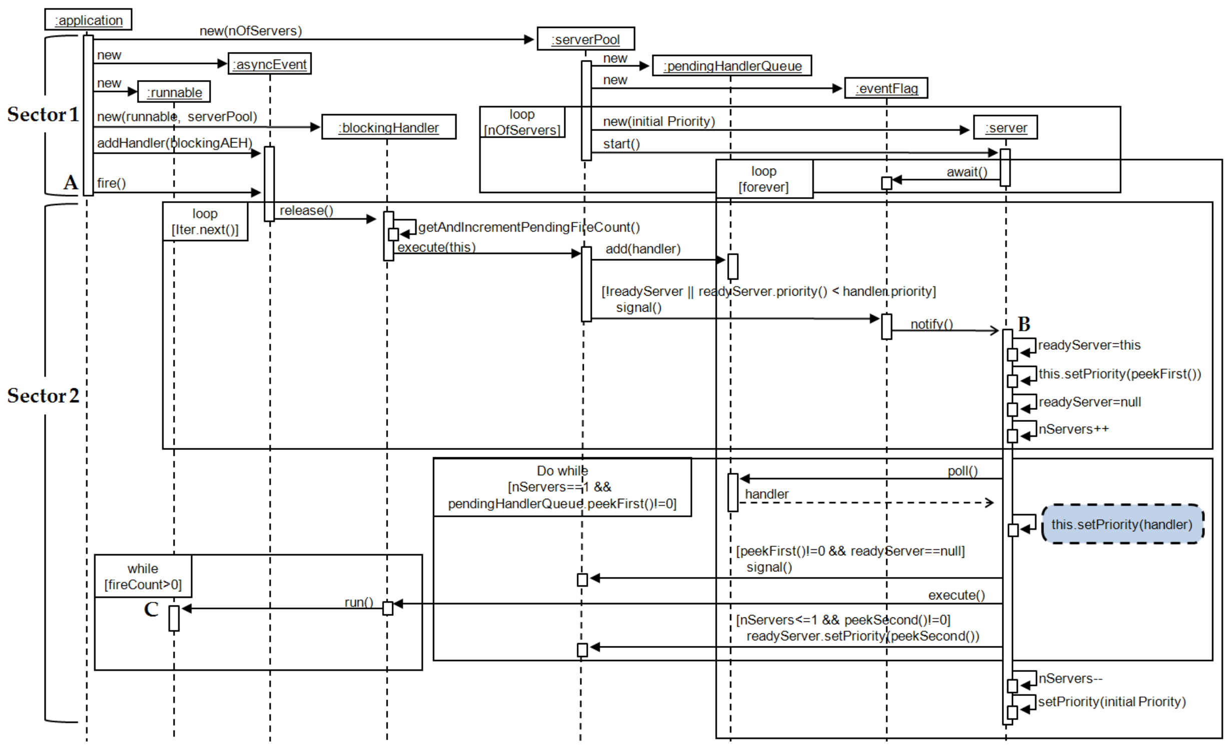


Figure 4.5: Blocking AEH Sequence Diagram

constructs the 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 initialised. 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 a handler as a parameter when the handler is constructed, forcing that the handler is served by one of the servers in the pool. Creating an `AsyncEvent` and attaching the constructed handler to the event 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's initial priority must be equal to or greater than the highest priority of all the handlers that the pool manages in order to avoid priority inversion. The notified servers will later adjust their priorities to the appropriate values accordingly.

Sector 2

This section represents the implementation-specific part of the AEH. When an `AsyncEvent` is fired, all the attached `AsyncEventHandlers` are released in a priority-ordered manner and put in the pending handler queue. After the queuing, one of the servers in the pool is notified when there is no ready server or the priority of the ready server is lower than the priority of the released handlers. This iteration continues until all the `AsyncEventHandlers` attached to the `AsyncEvent` are released. The notified servers perform a set of operations before they actually execute handlers in order for them to reflect the current state of the AEH subsystem. The set of operations is shown in the lifeline of the `server` in Figure 4.5. A notified server first sets itself as the ready server and then adjusts its priority to the first handler's priority in the queue. Here normally the server is preempted if there are servers with higher priorities. Otherwise it continues running and unregisters itself from the ready server. Note that, if there is the ready server and its priority is lower than that of the released handler, the priority of the ready server is increased to that of the released handler. This late binding behaviour is not depicted in the figure for clarity. The ready server then increases an integer variable, `nServers`, which represents the number of active servers in the AEH subsystem. The variable will be used later to determine whether servers should continue to execute or to release the CPU. The server now goes in the `Do-While` loop that constantly takes the first handler from the queue and executes it when the number of active servers is one and there is at least one pending handler in the queue. Inside the loop it first adjusts its priority to the first handler's in the queue after invoking the `poll()` operation on the queue, which returns the first handler after removing it. The `setPriority()` method inside a dashed circle in the figure is performed in a synchronised block on the server class, that gives the current server mutually exclusive access to the class. This is one of the possible ways that enable the blocking AEH model to prevent the multiple server switching phenomenon observed in the Java RTS AEH. This is due to the fact that in many operating systems including POSIX, it is generally the case that, if a thread holding a lock lowers its priority, it is recommended to be placed at the head of the thread run queue at its new priority[3]. After adjusting its priority, the server notifies another server if and only if there are pending handlers and the ready server is null. The server now finally executes the removed handler. After the completion of the handler, the server is allowed to loop as long as the condition in the `Do-While` loop is satisfied. Before the running server goes back to the start of the loop, it adjusts the ready server's priority to the priority of the second handler in the queue. This is done to prevent the ready server's preemption of the currently running server and let the running server continue the execution. If the `Do-While` condition is not satisfied, it decreases the variable, `nServer`, and sets its priority back to the initial value. Lastly the server goes back to the waiting state where it sleeps until it gets notified. The blocking AEH implementation have its own set of rules to achieve Requirement 1,2, and 3 presented in Section 4.1. It invokes server threads only when necessary and if the current running server does not block, it is allowed to continue to execute with another handler, reducing the number of active server threads on average.

4.3 Non-Blocking AEH

The non-blocking AEH model proposed to take advantage of non-blocking handlers to achieve a more lightweight AEH model. This stems from the fact that if a handler will not self-suspend, it is not required to have another server ready as a precautionary measure in the case of the handler's blocking, reducing the number of active servers on average. The following set of rules defines the run-time behaviour of the non-blocking AEH model:

1. A server is invoked if and only if there is no running (or ready) server or its priority is lower than that of the released handler.

[3]The RTSJ is not very clear how a schedulable should behave when it holds a lock and changes its priority at the same time. However, as servers, strictly speaking, are not RTSJ schedulable objects, they are allowed to have these semantics.

2. The currently running server is allowed to execute another handler after the completion of the current handler if and only if the highest priority among preempted server threads is lower than that of the first handler in the pending handler queue (i.e. the highest priority pending handler). Otherwise it yields the CPU.

This set of rules is to allow non-blocking handlers to be used with the dynamic 1:N mapping while meeting Requirement 1, 2, and 3 in Section 4.1. According to the critical sequence for the dynamic 1:N mapping model, the only handler releasing sequence that requires an additional server thread is when a higher priority handler is released while a lower priority handler is being executed for non-blocking handlers. The first rule is to utilise this critical sequence for non-blocking handlers. The second rule is to prevent the MSSP between the current running server and preempted servers. Consider the case where the current running server thread has preempted the previous running server and therefore there is a preempted server in the thread run queue at a lower priority than that of the current server. If a handler with a lower priority than that of the preempted thread is released at this instance and the current running server is allowed to execute further, there will be a MSSP entailed as the preempted server thread becomes to have a higher priority than the current running server when it changes it priority to that of the pending handler in the queue. In order to prevent this, the current server thread yields the CPU if the highest priority pending handler is lower than the highest preempted server thread. Based on this set of rules, the non-blocking AEH UPPAAL automata model is constructed, which is presented in Section 4.3.1. The actual implementation of the non-blocking AEH model is presented in Section 4.2.2. Note that, in the non-blocking AEH model, there is an additional data structure, called priorityStack which keeps track of all the priorities of the servers currently active (i.e. running with a handler bound to it). A running server adds its priority to and removes from the top of priorityStack in a LIFO manner. When the running server is preempted, the preempting server adds its priority to the top and the preempted server's priority becomes the second element of the stack. Therefore the first element, the *top*, of the stack always represents the highest priority at which the current server is running. The size of the array is bounded to the number of servers in the pool. The data structure is designed to implement the first rule for non-blocking AEH. Also note that priorityStack cannot be used with the blocking AEH model as blocking handlers do not allow the stack to store priority values in a LIFO manner as blocking happens randomly.

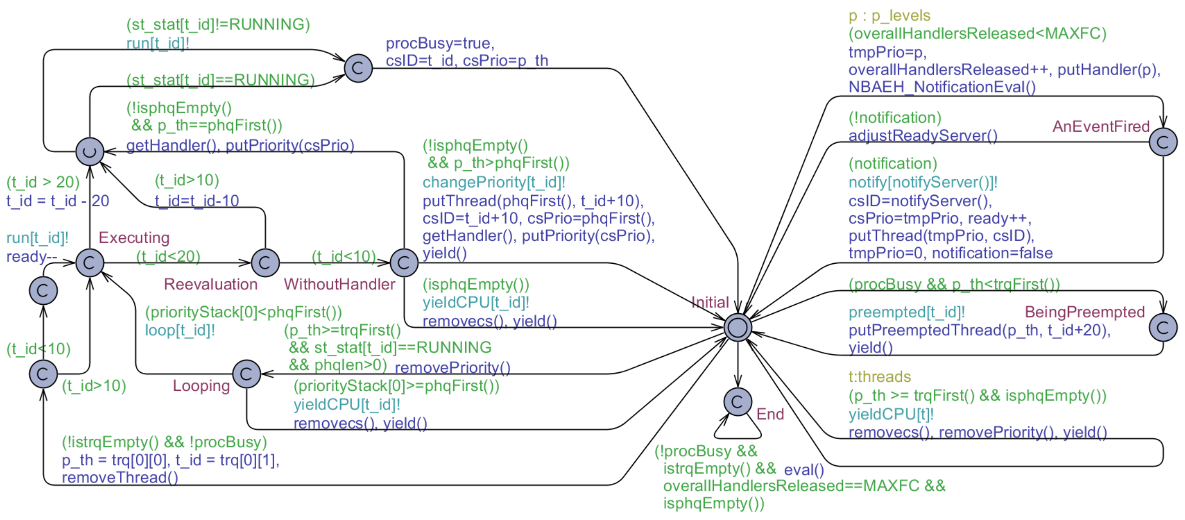


Figure 4.6: `Scheduler` Automaton for Non-Blocking AEH model

4.3.1 UPPAAL Automata Model for Non-Blocking AEH

Again here, the same `ServerThread` automata have been used for this model. Therefore only the `Scheduler` automaton used in the non-blocking AEH model is presented.

The Scheduler Automaton

The automaton in Figure 4.6 represents the base scheduler of the RTSJ (i.e. `Priority-Scheduler`) for the blocking AEH. The automaton also has six outgoing transitions from the initial location, synchronising with the `ServerThread` automata. Some transitions are identical with ones in the `Scheduler` automaton for blocking AEH shown in Figure 4.4. Hence we only discuss the different transitions from the previous ones as follows.

1. The transition from the location `AnEventFired` to `Initial`: In the Non-Blocking AEH model, a server in the pool is notified only when there is no server executing, the current server's priority is lower than that of the released handler, or there is no ready server, evaluated by invoking `NBAEH_NotificationEval()`. This behaviour is depicted in the transition.

2. The transition from the location `Looping` to `Initial`: When the current server is looping, it yields the CPU if the `priorityStack[0]` is equal to or greater than the priority of the first handler in the queue. This depicts the situation where there is a preempted server with a priority level which will be equal to or greater than that of the current server after it changes its priority to that of the first handler in the

	PRIORITY_LEVELS	MAXFC	Max STs (i)	Min STs (i)
Non-Blocking AEH	3	2	2	1
		3	3	1
		4	3	1
		5	3	1
	4	3	3	1
		4	4	1
		5	4	1
		6	4	1
	5	4	4	1
		5	5	1
		6	5	1
	6	5	5	1
		6	6	1

Table 4.2: Numbers of Server Threads and Non-Blocking AEH

queue. If the current server does not yield the CPU and continues the execution with the first handler in the queue, there will be a MSSP as the current server is to be preempted by a waiting server with a equal or higher priority. Therefore the current server yields the CPU.

3. The location `WithHandler` is absent: This is because this model no longer notifies another server in case of the currently running server's blocking.

4. The operations `put` and `removePriority()`: The two operations are performed each time a server starts and finishes the execution of a handler, respectively, to enable the AEH subsystem to keep track of the priority levels being executed.

Formal Analysis of the Non-Blocking AEH model

The blocking AEH model is simulated and verified using the same set of the properties that are applied for the Java RTS and blocking AEH models. The Table 4.2 shows the verification results of the least upper and lower bounds of servers required as the number of active handlers increases. The least upper bound of server threads required in the non-blocking AEH model is now completely dependent on the number of priority levels (not on the number of released handlers) and the least lower bound of servers required

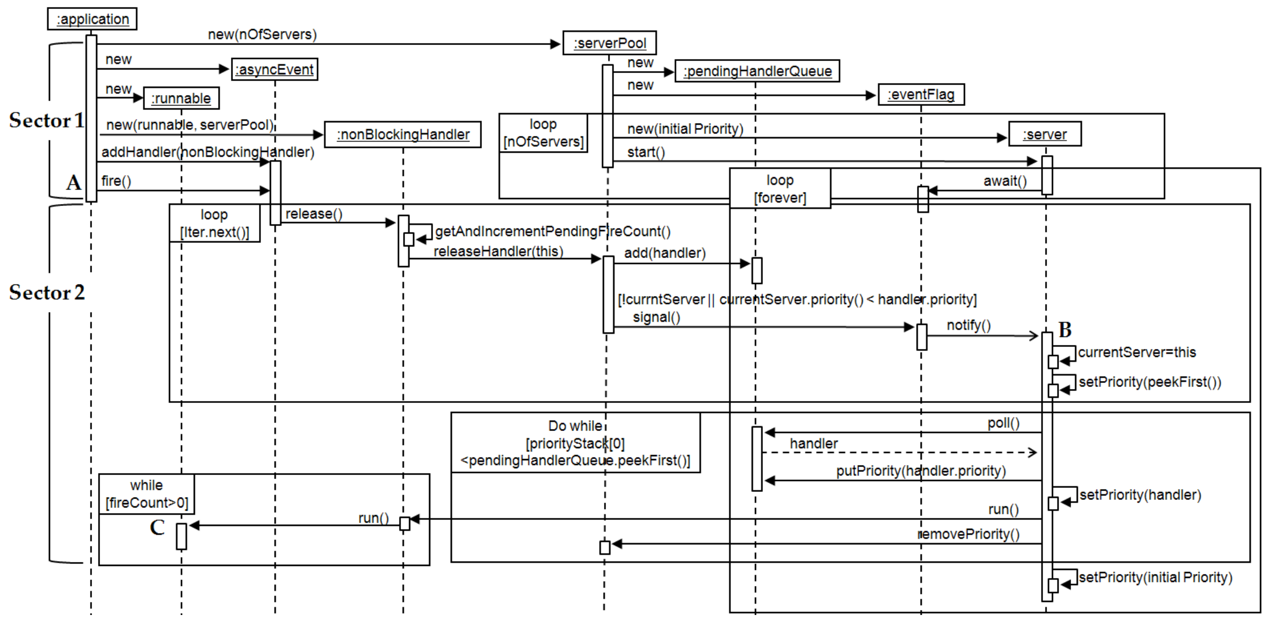


Figure 4.7: Non-Blocking AEH Sequence Diagram

is 1 regardless of how many handlers are released. This is possible as the model assumes handlers would not block and hence does not require anther server to be ready in the case of the currently running server's blocking. The non-blocking model is also capable of handling the MSSP as discussed earlier[4].

4.3.2 Non-Blocking AEH Implementation

Figure 4.7 shows the interaction between relevant objects in the blocking AEH model using a sequence diagram as before. In the figure there are two separate sectors, `Sector 1` and `Sector 2`, each of which depicts application-specific and implementation-specific part of the model, respectively. When an event is fired, a server in the pool is notified only if there are no active servers (i.e. `!currentServer`) or the priority the active server is lower than that of the released handler. A notified server sets itself as the current server and goes in the loop (i.e. `loop[forever]`). In the loop the server polls the first handler in the queue, puts the handler's priority value in the `priorityStack`, and adjusts its priority to that of the removed handler. Then the server executes it without notifying another server regardless of the presence of pending handlers in the queue. This is due to the fact that this model is for non-blocking handlers and hence does not need to have

[4]The source code also contains the symbolic trace that shows how the non-blocking AEH model avoids the MSSP. (NonBlockingAEHModel_Avoiding_MSSP.xtr)

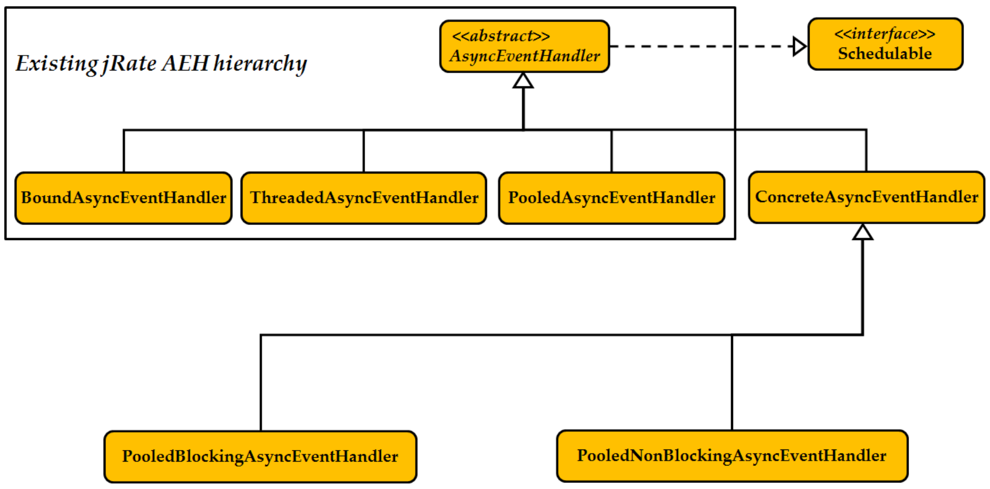

Existing jRate AEH hierarchy
<<abstract>>
AsyncEventHandler
<<interface>>
Schedulable
BoundAsyncEventHandler
ThreadedAsyncEventHandler
PooledAsyncEventHandler
ConcreteAsyncEventHandler
PooledBlockingAsyncEventHandler
PooledNonBlockingAsyncEventHandler

Figure 4.8: jRate AEH Hierarchy and New Handler Classes

another server waiting at the priority of the first handler in the case of the currently running server's blocking. After the completion of the handler it removes the handler's priority from `priorityStack` and continues to execute only when there are one or more pending handlers in the queue and the highest priority from the `priorityStack[0]` is lower than the priority of the first handler in the queue. The reason for the currently running server's releasing the CPU when the priority of the first handler in the queue is lower than the highest priority among active servers in the AEH subsystem is again to avoid the MSSP. Assume that there is a server executing a handler at the priority 5. The array, therefore, holds the priority 5 at `priorityStack[0]`. While the execution of the handler an event, which releases `H6`, `H4`, `H3`, is fired. A server is therefore notified and it preempts the currently running server. It then takes out the first handler, `H6`, from the queue and changes its priority to 6, continuing the execution. After the completion of the handler, it releases the CPU because the highest priority from the array, which is 5 at the time, is greater than the priority the first handler in the queue, which is 4. If the current server was allowed to continue the execution, the server will lower the priority to 4 and it will be immediately preempted by the waiting server at the priority 5. This preemption would recur until all the handlers are served. Therefore the current server yields the CPU. The yielding server sets its priority back to the initial value and goes back to the waiting state where it sleeps until it gets notified.

4.4 Performance Evaluation of the New Models

In this section, it is shown that how the two new AEH models perform compared to other AEH mapping models by carrying out two performance tests. The first test determines the dispatch latency of the server and handler when a single handler is released. The second test investigates the performance of the multiple-handler completion latency when there is more than one handler simultaneously released. However it is expected for the two new AEH models to incur higher overheads for the first test (a single handler release) as the algorithms of the server threads in the models are adaptive to the surrounding environment and hence complex in an attempt to reduce the number of server threads on average. The use of the models is justified by their efficiency when there are many handlers released simultaneously in the second test. To perform the tests, we have implemented the proposed AEH models in jRate running on top of an open-source RTOS, MaRTE OS [53], which allows us to emulate RTSJ-compliant applications on the RTOS using a Linux environment. Figure 4.8 shows the relationships between the existing jRate AEH classes and handler classes that are served by the new AEH models discussed in the previous section. The classes outside the existing jRate subsystem hierarchy in the figure represent the handler classes that use the new models implemented on the platform and the following briefly discusses each class.

- `AsyncEventHandler`: The class encapsulates the code that gets executed when an `AsyncEvent` occurs. It implements the `Schedulable` interface and is defined as an abstract class[5]. This is the parent class of all the classes each of which implement the respective AEH models.
- `BoundAsyncEventHandler`: This class implements the `AsyncEventHandler` class that has a single server dedicated to it.
- `ThreadedAsyncEventHandler`: It creates a server at run-time when required like the RI does (i.e. when a handler is released).
- `PooledAsyncEventHandler`: This implements the jRate AEH model discussed in Section 3.1, which uses the Leader/Followers design pattern.

[5]Earlier version of the RTSJ defines the `AsyncEventHandler` as abstract and jRate was implemented according to the earlier version of the RTSJ. However it is changed to be a concrete class in the most recent RTSJ, and the `ThreadedAsyncEventHandler` and the `ConcereteAsyncEventHandler` in Figure 4.8 implement the concrete version of the `AsyncEventHandler`.

- `ConcreteAsyncEventHandler`: This class behaves the same as the `ThreadedAsyncEventHandler` above. This is our own version of the `ThreadedAsyncEventHandler` in jRate and the implementation for this class slightly differs from the `ThreadedAsyncEventHandler`.
- `PooledBlockingAsyncEventHandler`: The class implements the handler class that uses the blocking AEH model and therefore the handlers which are derived from this class may block.
- `PooledNonBlockingAsyncEventHandler`: The class implements the handler class that uses the non-blocking AEH model and therefore the handlers which are derived from this class may not block.

Some of the AEH algorithms discussed in Section 3.1 have not been implemented for the following reasons.

1. The algorithms used in the RI and OVM are the simplest ones that just create a server as needed or allocate a single server per handler. The `ThreadedAsyncEventHandler` and `ConcreteAsyncEventHandler` classes take the same approach as the RI and the `BoundAsyncEventHandler` class uses the same AEH implementation technique as OVM does.
2. The algorithm used in Jamaica allocates a server per priority level. This seems to be using fewer servers. However this is not entirely true in the sense that for the blocking handlers the algorithm still requires the same number of servers to prevent priority inversion and therefore it would need a considerable amount of run-time server creation and destruction overhead, and for the non-blocking handlers it does not allow servers to execute multiple handlers with different priorities. Consequently it is not hard to notice that the algorithm is not optimal in terms of serving a set of handlers using the least possible number of servers.
3. The algorithm used in the proposed blocking AEH model essentially extends and improves the AEH algorithm used in Java RTS (i.e. preventing the multiple server switching phenomenon).

Note that, the current RTSJ neither specifies how two different AEH algorithms should be combined in the implementation nor allows the programmer to construct his/her own AEH algorithm. In order to implement more than one AEH algorithm in the same RTSJ implementation, the characteristics of the different implementation have to be reflected in

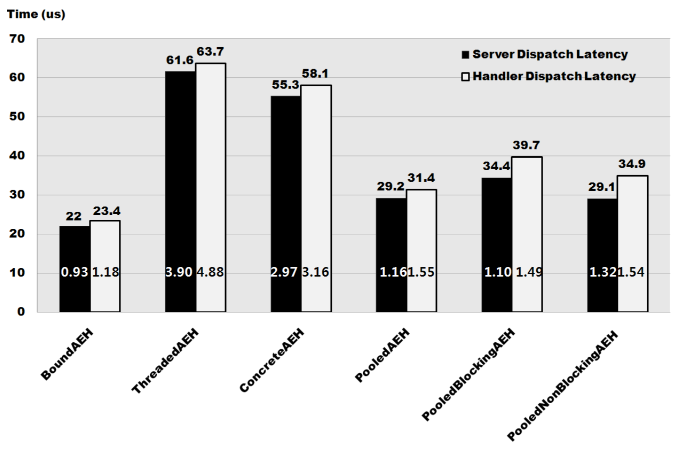


Figure 4.9: Dispatch Latency

the handler class hierarchy. In the case of the blocking and non-blocking AEH models, they are constructed in separate classes each of which uses its own AEH algorithm (i.e. released `PooledNonBlockingHandlers` are stored in a `NonBlockingPriorityQueue` and are executed by `NonBlockingServerThreads` in a `NonBlockingServerPool`, while released `PooledBlockingHandlers` are stored in a `BlockingPriorityQueue` and are executed by `BlockingServerThreads` in a `BlockingServerPool`). In the following sub-sections, we present and analyse the results from the two tests that are performed on the AEH models discussed above.

4.4.1 AEH Dispatch Latency Test

This test measures the dispatch latency of the various classes of `AsyncEventHandler` in Figure 4.8. There are two types of dispatch latency. One is the server dispatch latency that represents the time taken from an `AsyncEvent` being fired, marked as A in Figures 4.5 and 4.7, to its potential server being released, marked as B in the figures. The other is the handler dispatch latency, which is the time taken from an `AsyncEvent` being fired (A in the figures) to its associated `AsyncEventHandler` starting to execute (marked as C in the figures). This test therefore shows how long it takes for a server in the AEH models to be invoked (server dispatch latency) after a handler being released and to start to execute the released handler (handler dispatch latency).

Test Settings

To measure the two dispatch latencies incurred in the models discussed above, we performed a test with an `AsyncEvent` being fired 1,000 times in a discrete manner to ensure

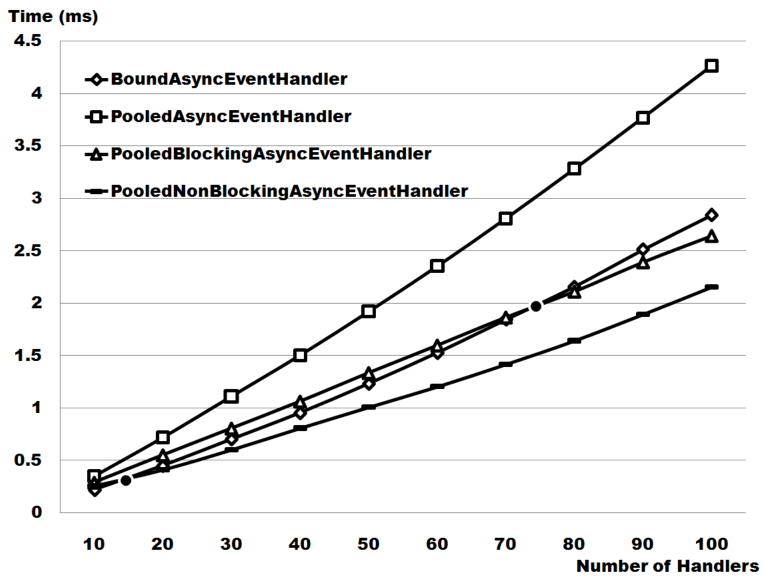


Figure 4.10: Multiple-handler Completion Latency Test with 10 to 100 Handlers

that each event firing causes a complete execution cycle, where an event-firing thread may fire the event only after the previously released handler is completed. The `AsyncEvent` has one associated `AsyncEventHandler` that will be released when the `AsyncEvent` is fired. 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 areas as their current memory areas.

Test Results and Analysis

Figure 4.9 illustrates the average dispatch latency of the various models in jRate and the two new AEH models. The standard deviations for the tested AEH models are also presented in the measurements of latencies (bars) in Figure 4.9. The `BoundAsyncEvent-Handler` class incurs the lowest average dispatch latency for both server and handler dispatch. The `Threaded` and `ConcreteAsyncEventHandler` require the highest average dispatch latency as they entail the run-time server creation overhead whenever events are fired. The AEH model used in jRate and the two new AEH models produce the similar amount of the average dispatch latency that is placed in between the two average dispatch latency for `BoundAsyncEventHandler` and `ThreadedAsyncEventHandler`. In the two new AEH models however the time difference between the server and the handler dispatch latency is bigger and this latency represents the time taken for servers to update information according to the current state of the AEH subsystem before it actually executes

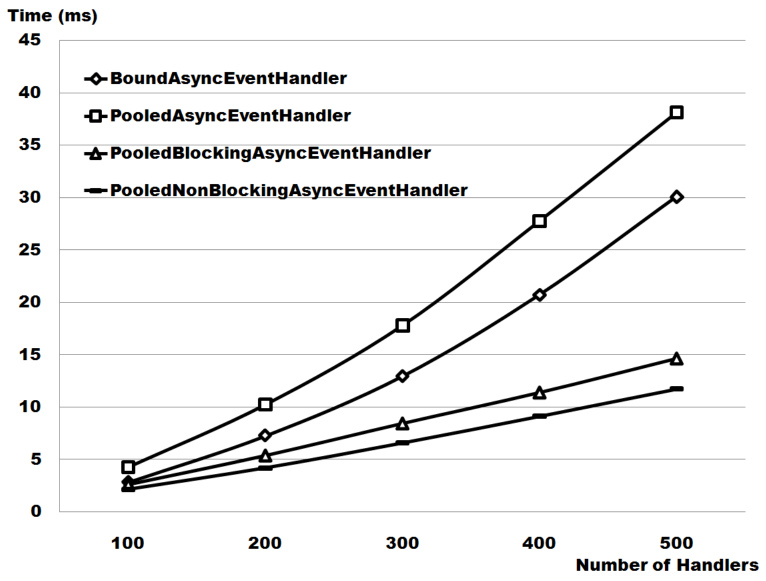


Figure 4.11: Multiple-handler Completion Latency Test with 100 to 500 Handlers

the released handler. Therefore the blocking and non-blocking AEH models incur slightly higher server and handler dispatch latencies than those of the dynamic 1:1 mapping model in jRate. This represents the fact that a server thread in the two new AEH models is computationally heavier to cater for their advanced algorithms which are to reduce the number of server threads on average. Although the results from this test do not explicitly show the benefits of using the new models as they have larger dispatch latency for both server and handler, they will become apparent in the next test with multiple released handlers as the handler dispatch latency from A to C in Figure 4.5 and 4.7 will not be recurring while other models will require the full length of the handler dispatch latency for each released handler.

4.4.2 Multiple-handler Completion Latency Test

This test measures multiple-handler completion latency, the time taken from various numbers of handlers being released to the last handler being completed. Therefore, this test measures the time taken to complete a certain number of handlers. We performed the test with the four AEH models, `BoundAsyncEventHandler`, `PooledAsyncEventHandler`, `PooledBlockingAsyncEventHandler` and `PooledNonBlockingAsyncEventHandler`.

Test Settings

Various numbers of handlers, from 10 to 500, are created for each AEH model per measurement and they are attached to a single event. They all have the same runnable object to make sure that they are assigned the exactly same logic to execute, including the execution time. 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 released handler being completed. Again, to avoid the interference of the garbage collector while performing the test, the real-time thread that fires the event and the servers that execute the released handlers use scoped memory areas as their current memory areas.

Test Results and Analysis

Figures 4.10 and 4.11 show the trends of the completion latencies of various numbers of released handlers for the tested AEH models. Figure 4.10 presents the trends of the completion latencies for relatively smaller numbers (10 to 100) of released handlers and Figure 4.11 demonstrates the trends for larger numbers (from 100 to 500) of released handlers. Obviously, the latencies increase as the number of released handlers increases. The `PooledAsyncEventHandler` produces the highest completion latency. This is because each time a `PooledAsyncEventHandler` is released, the run-time dynamically bounds a waiting server to it. This latency is incurred for each release of `PooledAsyncEventHandler`. The `BoundAsyncEventHandler` eliminates the latency by having a dedicated server for each handler. The trend of the `BoundAsyncEventHandler` in the figures reflects this fact. The trends of the multiple-handler completion latencies for the `PooledAsyncEventHandler` and the `BoundAsyncEventHandler` grows linearly until the number of released handlers reaches approximately 200, after which the trends increase exponentially. This illustrates that the platform on which the tests are performed can support a maximum of roughly 200 concurrent threads, after which the throughput of the system degrades. It is also shown that the increasing rate of the `BoundAsyncEventHandler` is greater than the two new AEH models. At the start, with 10 handlers, the `BoundAsyncEventHandler` incurs the lowest latency. However as the number of handlers increases, it incurs more overhead to complete than the two new models. Between 10 and 20 handlers, the performance of the `PooledNonBlockingAsyncEventHandler` becomes better and the performance gap is growing afterwards. For the case of the `PooledBlockingAsyncEventHandler`, it is between 70 and 80 handlers with the blocking AEH model producing better performance.

	100	200	300	400	500
Bound AEH	28.3	36.3	43.1	51.8	60.1
jRate AEH	42.6	51.2	59.2	69.3	76.2
Blocking AEH	26.4	27.0	28.2	28.5	29.3
Non-blocking AEH	21.5	21.0	21.9	22.8	23.4

Table 4.3: Per Handler Overhead as the Number of Released Handlers Increases

The two points where the two new models becomes to outperform the `BoundAsync-EventHandler` model are marked with the black-colored circles in Figure 4.10. Figure 4.11 clearly shows that the performance differences become more apparent as the number of released handlers increases. When the number of handlers reaches 500, the two new models exhibit more than 50% better performance. This performance gain of the two new models is achieved by minimising the number of concurrently active servers. Regardless of the number of released handlers, the `PooledBlockingAsyncEventHandler` requires 2 servers and `PooledNonBlockingAsyncEventHandler` needs 1 server, for the scenario of this test. This is because all the handlers are attached to a single event and therefore when the event is fired, the associated handlers are released in a priority-decreasing-ordered manner. According to the notification rule of the two new models for this scenario, only 2 and 1 servers are required, respectively. Table 4.3 shows per-handler-overhead as the number of released handlers increases (i.e. overall overhead divided by the number of released handlers). The per-handler-overheads for bound and jRate AEH grow as the number of released handlers increase. This clearly represents the fact that as the number of threads in the system increases, the overall performance of the system decreases. When the number of released handlers reaches 500, the per-handler overheads for them incur more than 50% higher overheads than that of the case when there is only one released handler. On the other hand, the blocking and non-blocking AEH models produce similar per-handler-overhead regardless of the number of released handlers. They require only 38% and 30% of the per-handler-overhead, respectively, compared to the AEH model in jRate with 500 simultaneously released handlers. Therefore the overall utilisation with 500 simultaneously released handlers is reduced by 30.1% which is derived by combining the per-handler-latencies of the two new models divided by that of the AEH model in jRate.

4.5 Schedulability Analysis for the AEH Models

This section shows how the standard response time equation can be extended to cater for the blocking and non-blocking models. As discussed in Section 2.3.2, the standard response time equation for an arbitrary task i, t_i, in a fixed-priority system is as the following [2]:

$$R_i = C_i + B_i + \sum_{j \in hep(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \tag{4.3}$$

where R_i, C_i, B_i, and T_i represent the worst-case response time, the worst-case computation time, the worst-case blocking time and the period of t_i, respectively. Also $hep(i)$ represents the set of tasks with a higher or equal priority than/to i. To account for the extra delay suffered by a task i due to its own self-suspension and the suspension of higher priority tasks, the blocking (self-suspending) factor denoted as $b_i(ss)$ is typically included in the total blocking time of t_i, B_i. $b_i(ss)$ in a fixed-priority system can be determined using the following equation [45]:

$$b_i(ss) = mss_i + \sum_{j \in hep(i)} \min(C_j, mss_j) \tag{4.4}$$

where mss_i denotes the maximum self-suspension time of t_i. This is because the worst-case blocking time occurs if a lower or equal priority task is released when the self-suspension of a higher or equal priority just finishes. This may result in the whole computation time of the previous instance of the higher or equal priority task contributing the blocking time of a lower or equal priority task. Therefore the worst-case blocking time is the minimum of the computation time and the self-suspension time of a higher or equal priority task.

The context switch overhead can also by accounted for in the schedulability analysis by including the time spend for the two context switches at the start and completion of each task as part of the execution time of it [6, 45]. If the task is preempted and later resumed, the time spent for the two context switches can be accounted for in the same way; by including the context-switch time in the execution time of the preempting task. Assuming that cs denotes the context-switch time of the system (i.e. the maximum amount of time the system spends per context switch) and cs includes the time required to maintain the context of the tasks involved in the context switch as well as the time spent by the scheduler to service the event interrupt that triggers the context switch and to carry out the scheduling action at the context switch (e.g. clock handler). If no task ever self-suspends, the execution time of every task should be increased to $C_i + 2cs$, for i

= n, n-1, ..., 1. Therefore Equation 4.3 becomes:

$$R_i = 2cs + C_i + B_i + \sum_{j \in hep(i)} \left\lceil \frac{R_i}{T_j} \right\rceil (2cs + C_j) \tag{4.5}$$

The context-switch at the completion can be removed to obtain the response time of t_i, that exclude the time taken to context switch away from the CPU, by decreasing the first $2cs$ to cs. Also, if tasks self-suspend more than once per release, a task incur two more context switches each time it self-suspends. Therefore, if each task can self-suspend a maximum of K_i time after its execution starts, $2(K_i + 1)cs$ (or $(K_i + 1)cs$ to exclude the context-switch at the completion) should be added to the execution time C_i.

The schedulability analysis presented in Equation 4.3 and 4.5 based on the fact that each task has it own thread of control for execution. Therefore, the blocking AEH model, which allocates a single server thread to each handler in the worst case, can be applied directly to the analysis without modification. For the non-blocking AEH model, Equation 4.5 should be extended to cater for the 1:N mapping model, whose least upper bound of server threads is equal to the number of priority levels. This is due to the fact that the second $2cs$ may not incur if next handlers are executed by a single server thread. The following equation shows the extended schedulability analysis that provides an upper bound on context-switch times for the non-blocking AEH model:

$$R_i = 2cs + C_i + B_i + \sum_{j \in hep(i)} \left\lceil \frac{R_i}{T_j} \right\rceil (need(j+1)2cs + C_j) \tag{4.6}$$

where $need$ is a 0/1 function: $need(j + 1) = 0$ if the priority of t_j's previous task in the task set from the highest to the lowest priority task is equal to that of t_j. This represents that if the priority of the previous checked task is equal to that of the currently being checked task, the server thread used to execute the previous task will still be used to execute the current task and hence no context switches will be incurred. Otherwise it gives the result 1. The non-blocking AEH model, therefore, reduces the overhead associated context-switch as shown in Equation 4.6, even in the worst case. This further accelerates overall performance of the system as fewer active server threads will reduce the scheduling overheads significantly – for example, the time taken to execute the clock handler can vary considerably, depending on the number of ready or blocked threads [6]. Note that the worst-case blocking time, B_i in Equation 4.6 does not include the self-suspension, $b_i(ss)$, as handlers are not allowed to block in this model.

4.6 Summary

This chapter has proposed and implemented efficient models for asynchronous event handling in the RTSJ. The underlying idea beneath the proposed models is to use as few servers as possible. This is to achieve the primary intention of the AEH in the RTSJ. The proposed models are designed to serve handlers with different characteristics, blocking and non-blocking. To validate the idea, the dynamic 1:N mapping model was discussed and emphasised for its efficiency and scalability. Based on the dynamic 1:N mapping model, the notion of critical sequences for blocking and non-blocking handlers is defined to state the condition under which the least upper bound of server threads is required. Each blocking handler requires a single server and non-blocking handlers collectively requires the number of servers as the number of priority levels. Based on this, the equations that calculate the least upper bound of servers required for both blocking and non-blocking handlers were derived for a given set of handlers. The blocking and non-blocking AEH implementations are constructed, based upon the dynamic 1:N mapping model and utilising the critical sequences. The blocking AEH model allocates a single server to each blocking handler and the non-blocking AEH implementation only invokes a server to cater for preemption. The 1:1 mapping model, such as bound asynchronous event handlers in the current RTSJ, always requires the least upper bound of servers unless handlers are released in a discrete manner. On the other hand, the blocking AEH model will require the least upper bound of servers if and only if a critical sequence occurs. The non-blocking AEH model further reduces the least upper bound of servers by taking advantage of non-blocking handlers. It requires the same number of servers as that of the priority levels in the system, even in the critical sequence. All the other handler releasing scenarios for the non-blocking AEH model require fewer servers than the least upper bound of servers in the critical sequence. Given the fact that the two implementations do not exhibit big differences in performance, requiring the least possible number of servers on average would justify the use of the non-blocking AEH model, as it will make an application more lightweight (if the application consists of only non-blocking handlers). Non-blocking handlers can also be served by the blocking AEH model, but not vice versa.

We also have discussed how the proposed models avoid the concurrency-related side effects such as priority inversion and the multiple server switching phenomenon while keeping the number of active servers in the AEH subsystem to a minimum. After implementing the AEH models, we ran a set of performance tests that measured the dispatch and multiple-handler completion latency of various AEH models. In the tests, the two new models have shown promising results that outperforms the other AEH implementations in jRate. The models have even shown better performance than the `BoundAsyncEventHandlers` after the point where there are more than 20 or 80 handlers in the non-blocking and the blocking models, respectively. Also the performance gap between them becomes bigger and bigger as the number of handlers increases. This clearly shows that allocating a dedicated server to each handler leads to a performance degradation even if a dedicated server per handler exhibit better performance for a smaller number of released handlers. This is somewhat surprising and interesting results in the sense that dynamically binding servers to handlers at run-time can result in better performance than allocating a dedicated server per handler. Our test results indicate that current AEH implementations of the RTSJ could be optimised to use as few servers as possible, incurring less overhead. The blocking and non-blocking AEH models are prime examples of such an optimised model and on average requires a smaller number of servers and context-switches. Note that the overheads measured in Figure 4.9 s not the worst-case observations. They are average overheads from the observations of 1000 times of a handler release. If a safe result is to be obtained for a hard real-time system, the worst-case overheads must be incorporated. However general practice suggests that this is not always done.

In order to apply schedulability analysis to the blocking and non-blocking AEH models, the standard worst-case response time analysis for fixed-priority systems was discussed. The blocking AEH model can directly be applied to the standard schedulability analysis without modification. On the other hand, the schedulability analysis has been extended to cater for the non-blocking AEH model. The extended analysis in Equation 4.6 showed how the non-blocking model reduces its overheads associated with context-switches. $2cs$ is incurred for every higher priority task in Equation 4.5, while $2cs$ is incurred for every higher or equal priority level in Equation 4.6.

The impact on the application programmers are minimal. They first have to decide which handler class to extend when designing a new asynchronous event handler. This will dictate which implementation approach the JVM will use. They also have to ensure that their handlers conform to what ever restrictions are required when using that approach (e.g. their handlers do not block).

Chapter 5

Applying Fixed-Priority Preemptive Scheduling with Preemption Threshold to AEH in the RTSJ

As shown in the previous chapter, the least upper bound of servers is entirely dependent upon two factors, the number of blocking handlers and the number of priority levels. If either of these numbers are reduced, the least upper bound of server threads can also be decreased. The number of blocking handlers in the system cannot be controlled at the implementation level as it is application-dependent. However the number of effective priority levels can be reduced by using fixed-priority preemptive scheduling with preemption threshold (FPPT) [9, 13, 52, 65] without jeopardising the schedulability of the existing task set. Therefore, periodic and non-blocking task sets can successfully be scheduled with FPPT as the least upper bound of server threads for non-blocking handlers depends on the number of priority levels. FPPT is primarily introduced to feasibly schedule a task set which is not schedulable either by fixed-priority preemptive scheduling (FPP) or fixed-priority non-preemptive scheduling (FPNP). In [65], the schedulability of a task set under two different scheduling policies, FPP and FPPT, is compared in a quantitative manner by using the notion of *breakdown utilisation* for randomly generated tasks. Period and utilisation of a task in the task set is chosen using a uniform distribution function.

The breakdown utilisation of a task set is defined as the associated utilisation of a task set at which a deadline is first missed when the computation time of each task in the task set is scaled by a factor. It is shown that FPPT improves 3-6% in schedulability compared to FPP, depending on the number of tasks (i.e. as the number of tasks increases, the improvement tends to decrease). FPPT is an extension of FPP in which each task has a supplementary priority called a *preemption threshold (or dispatch priority)*, in addition to its regular priority. This essentially results in a task having dual priorities. The regular priority of a task is used when the task is queued, but when the task gets the CPU its active priority is raised to its preemption threshold. The task keeps the raised priority until the end of its execution at which point the priority is lowered to its regular priority level. Therefore FPPT allows a task to prevent the preemptions from tasks with higher regular priorities up to its preemption threshold, meaning that a ready task can preempt the running task if and only if the ready task has a regular priority higher than the preemption threshold of the running task. As a result, FPPT reduces overheads due to fewer preemptions and context-switches by effectivly reducing the number of priority levels in the system, along with improved schedulability.

In this chapter, FPPT is applied to the non-blocking AEH model to further reduce the least upper bound of server threads even at the critical sequence for non-blocking handlers. In Section 5.1, schedulability analysis for FPPT is presented, which differs from that for FPP or FPNP. The schedulability analysis for FPPT is based on its own critical instance which is built on top of the notion of level-π_i busy interval. Section 5.2 explains various preemption threshold assignment algorithms, based on a task set with predefined regular priority. Section 5.3 presents and discusses the non-blocking AEH implementation extended to support the notion of preemption threshold. Following this, Section 5.4 discusses the impact of the extended non-blocking AEH implementation on the schedulability analysis and on the RTSJ. Lastly, Section 5.5 summarises this chapter.

5.1 Schedulability Analysis for FPPT

FPPT is based on a notion of level-π_i busy interval [43]. Assuming that $\mathbf{T}_i$ denotes a set of tasks with equal or higher priority than t_i, a level-π_i busy interval $(t_0,t]$ begins at an instant t_0 when (1) all jobs in $\mathbf{T}_i$ released before the instant have completed and (2) a job in $\mathbf{T}_i$ is released. The interval ends at the instant t after t_0 when all the jobs in $\mathbf{T}_i$ released since t_0 are complete. Therefore in the interval $(t_0,t]$, the CPU is busy all the time executing jobs with priorities π_i or higher, all the jobs executed in the busy interval are released in the interval, and at the end of the interval there is no pending jobs to be executed afterwards. The reason why all jobs of a task in the first level-π_i busy interval must be checked to determine the worst case response time of the task is that the first job $t_{i,1}$ may no longer have the largest response time among all jobs in the task due to non-preemption introduced in FPPT. This is because the critical instant for FPP no longer holds for FPPT due to the non-preemption. A critical instant of T_i is the time at which the release of the task will produce the largest response time [44, 45]. In a fixed-priority uniprocessor system where every task completes before the next release, a critical instance of a task occurs when it is released at the same time with every higher priority task. The critical instant for FPPT is defined in [65] such that

1. A job of each task with a higher or equal regular priority arrives at the same time (i.e. the FPP critical instant), and
2. A task A may be blocked by another task B with a lower regular priority, and a higher or equal preemption threshold than the regular priority of A. That is, the task B contributing the maximum blocking time has just started before the first critical instant in the level-π_i busy interval.

In addition to the FPPT critical instant, the following two properties are also shown to be true such that [65]:

- **Property 1**: There does not exist a universal critical instant for the whole task set in FPPT except when FPPT degrades to FPP.
- **Property 2**: FPPT is robust under the FPPT critical instant.

Property 1 represent the fact that the worst case scenario for each task in FPPT may not occur at the same time except when it degrades to FPP. In other words, under the worst case scenario for a task, it is not necessary to check whether all other tasks exhibit their worst case scenarios because their worst case scenarios may not exist at the same time. Property 2 can be rephrased such that if a task set $\mathbf{S}$ is schedulable by FPPT, then task set $\mathbf{S}'$ such that $\mathbf{S}' \subset \mathbf{S}$ is also schedulable.

Based on the critical instant and the level-π_i busy interval of FPPT, the following formulae 5.1 to 5.5 [52] are defined for the correct schedulability analysis for FPPT. A task's regular priority and preemption threshold are denoted by a pair of numbers (π_i, γ_i), where π_i denotes the regular priority and γ_i indicates the preemption threshold of t_i. By default, $\pi_i = i$. For a task set with n periodic tasks, the regular priority assignment is denoted as $\pi = \{\pi_n, \pi_{n-1}, ..., \pi_1\}$ and the preemption threshold assignment is expressed as $\gamma = \{\gamma_n, \gamma_{n-1}, ..., \gamma_1\}$. The preemption threshold γ_i of task t_i is therefore between i and n inclusively by definition. The greater the number, the higher the priority.

$$B_i = \max_{\pi_k < \pi_i \leq \gamma_k} C_k \tag{5.1}$$

$$W_i = B_i + \sum_{j \in hep(i)} \left\lceil \frac{W_i}{T_j} \right\rceil C_j \tag{5.2}$$

$$S_{i,k} = B_i + (k-1)C_i + \sum_{j \in hep(i)} \left(1 + \left\lfloor \frac{S_{i,k}}{T_j} \right\rfloor \right) C_j \tag{5.3}$$

$$F_{i,k} = S_{i,k} + C_i + \sum_{\forall j, \pi_j > \gamma_i} \left(\left\lceil \frac{F_{i,k}}{T_j} \right\rceil - \left(1 + \left\lfloor \frac{S_{i,k}}{T_j} \right\rfloor \right) \right) C_j \tag{5.4}$$

$$R_i = \max_{j=0,1,...,\left\lfloor \frac{W_i}{T_i} \right\rfloor} (F_{i,j} - (j-1)T_i) \tag{5.5}$$

Formula 5.1 computes the blocking time of t_i, B_i, from tasks with a lower regular priority, and an equal or a higher preemption threshold. The blocking time is the maximum execution time of such tasks, C_k. Formula 5.2 computes the length of the in-phase level-π_i busy interval, W_i, iteratively starting from $W_i^1 = \sum_{j \in hep(i)} C_j + B_i$ until $W_i^{l+1} = W_i^l$ for some $l \geq 1$, where $hep(i)$ represents the set of tasks with an equal or higher priority to/than t_i and T_i denotes the period of t_i. The solution W_i^l is the length of the level-π_i busy interval. Formula 5.3 computes the starting time of $t_{i,k}$, $S_{i,k}$, where $t_{i,k}$ denotes the kth job of t_i in the level-π_i busy interval. Before $t_{i,k}$ can start to run, all jobs of tasks with a higher or equal priority ready over $[0,\ S_{i,k}]$ must finish before $S_{i,k}$, therefore the first k jobs of t_i in addition to the blocking task with a lower regular priority. This formula is also solved iteratively, starting from $S_{i,k}^1 = (k-1)C_i + \sum_{j \in hep(i)} C_j + B_i$, where $hep(i)$ represents the set of tasks with an equal or higher priority to/than t_i. Formula 5.4 computes the finishing time of $t_{i,k}$, $F_{i,k}$. After $t_{i,k}$ starts to execute, its priority increases to its preemption threshold, and therefore t_i can only be preempted by a task t_j ready over $(S_{i,k},\ F_{i,k})$ where $\pi_j > \gamma_i$. Hence $F_{i,k}$ is equal to its starting time plus its computation time and all the computation times of tasks with higher regular priorities than γ_i with jobs ready over $(S_{i,k},\ F_{i,k})$. At time $S_{i,k}$, no jobs of task t_j with $\pi_j > \gamma_i$ is ready, therefore the number of jobs of task t_j ready before $S_{i,k}$ is equal to $\left\lceil \frac{S_{i,k}}{T_k} \right\rceil$. The iteration starts from $F_{i,k}^1 = S_{i,k} + C_i$. Formula 5.5 computes the worst case response time of t_i, R_i, within the in-phase level-π_i busy interval. A task set $\{t_i | 1 < i < n\}$ is schedulable by FPPT if and only if $\forall i, R_i < D_i$, where D_i represents the deadline of t_i.

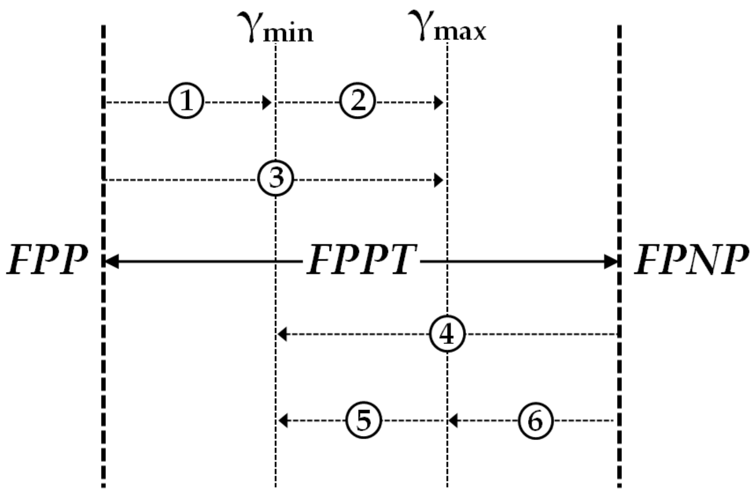


Figure 5.1: Preemption Threshold Assignment Algorithms in FPPT

5.2 Preemption Threshold Assignment

The schedulability analysis in Section 5.1 assumes a preemption threshold assignment already exists. In practice however, normally, regular priorities are solely defined by using a priority assignment algorithm for FPP (e.g. rate, deadline monotonic assignments, etc). In this section, various preemption threshold assignment algorithms are presented, based on a task set with predefined regular priority. In [9], it is formally proved that there may be none or more valid assignments for a task set to be schedulable by FTTP and all the valid assignments are delimited by two special assignments, the minimal assignment $\gamma_{min} = \{\gamma_{min_1}, \gamma_{min_2}, ..., \gamma_{min_n}\}$ and the maximal assignment $\gamma_{max} = \{\gamma_{max_1}, \gamma_{max_2}, ..., \gamma_{max_n}\}$ where $\gamma_{min_i} \leq \gamma_{max_i}$, for $1 \leq i \leq n$. In other words, for any valid assignment $\gamma = \{\gamma_1, \gamma_2, ..., \gamma_n\}$, it must satisfy that $\gamma_i \in [\gamma_{max_i}, \gamma_{min_i}]$, for $1 \leq i \leq n$, which is denoted as $\gamma_{min} \preccurlyeq \gamma \preccurlyeq \gamma_{max}$. Note that some assignment delimited by γ_{min} and γ_{max} may not be valid. Therefore FPPT degrades to FPP when the preemption threshold of each task is equal to its regular priority (i.e. $\gamma_{min} = \{\forall i | \gamma_{min_i} = \pi_i\}$). Similarly, when all preemption thresholds are the same as the highest regular priority in a task set, FPPT upgrades to FPNP (i.e. $\gamma_{max} = \{\forall i | \gamma_{max_i} = \pi_n\}$). When a task set with predefined regular priority is not schedulable by FPPT, neither γ_{min} nor γ_{max} exists.

There are six available preemption threshold assignments ① to ⑥ in the literature as shown in Figure 5.1. When a task set with predefined regular priority is schedulable by FPPT, its γ_{min} can be calculated by starting from either FPP or FPNP, indicated as algorithms ① [65] and ④ [9] respectively. If γ_{min} or γ_{max} is known, the other can be calculated, which is indicated by algorithms ② [66] and ⑤ [9] respectively. The algorithm ③ [9] computes γ_{max}, starting from FPP, by sequentially combine ① and ② as no algorithm that computes γ_{max} by starting directly from FPP exists. However the algorithm ⑥ [9] computes γ_{max} by starting directly from FPNP. The validity of the algorithms ① to ⑥ is formally proved in [9].

In order to minimise the number of priority levels for a task set, the maximal assignment γ_{max} should be used as any valid assignment $\gamma \preccurlyeq \gamma_{max}$. A task will boost its priority to preemption threshold as soon as it gets the CPU and will not be preempted by a task with a higher regular priority task, effectively reducing the number of running priority levels by preventing preemptions. When all preemption thresholds are the same as the highest regular priority in a task set, FPPT upgrades to FPNP (i.e. $\gamma_{max} = \{\forall i | \gamma_{max_i} = \pi_n\}$), effectively having only one running priority level. Therefore we are interested in finding γ_{max}. The algorithms ③ and ⑥ generate the same maximal assignment γ_{max} for a given task set if γ_{max} exists. Note that the two algorithms do not need to start with a valid assignment (i.e. the given task set may not be schedulable with the current fixed-priority assignment). Listing 5.1 shows the pseudo-code [9] for the algorithm ③ which is the sequential combination of algorithms ① and ②.

The algorithms ① and ② are depicted on lines 1 to 11 and 12 to 21, respectively. ① first sets the initial values for the regular priorities and preemption thresholds by assuming the preemption threshold of each task is the same as its priority (presented on lines 1 to 4). Then a for loop starts to calculate the minimal preemption threshold for each task from the lowest regular priority task to the highest regular priority one, which corresponds to lines 5 to 11. For any task, whenever it cannot meet its deadline with the current regular priority and preemption threshold (WCRT(π_i, γ_i) computes R_i using Formulas 5.1 to 5.5), the algorithm increases the preemption threshold by one. If the increased preemption threshold is higher than the highest regular priority π_n, it returns false, meaning that there is no valid assignment that can make the task schedulable. Otherwise the algorithm repeats the increment and checks if the current task can meet its deadline, depicted on lines 6 to 10. Based on the result γ_{min} from ①, ② starts to calculate the maximal preemption threshold for each task from the highest regular priority task to the lowest regular priority one, which corresponds to lines 12 to 21. The algorithm increases the preemption threshold γ_i by one, while it is lower than the highest regular priority. The worst case response time

```
for(i = 1 to n) {
    $\pi_i$ = $i$;
    $\gamma_i$ = $\pi_i$;
}
for(i = 1 to n) {
    while(WCRT($\pi_i$,$\gamma_i$) > $D_i$) {
        $\gamma_i$ = $\gamma_i$ + 1;
        if($\gamma_i$ > $\pi_n$)
            return false;
    }
}
for(i = n to 1) {
    while($\gamma_i$ < $\pi_n$) {
        $\gamma_i$ = $\gamma_i$ + 1;
        k = $\gamma_i$
        if(WCRT($\pi_k$, $\gamma_k$) > $D_k$) {
            $\gamma_i$ = $\gamma_i$ - 1;
            break;
        }
    }
}
```

Listing 5.1: Maximal Assignment Algorithm from FPP

of the potentially affected task t_k is recalculated, where k is equal to the new preemption threshold γ_i. If t_k misses its deadline where $\pi_k = \gamma_i$, then the increasing the preemption threshold of γ_i is invalid. Then γ_i is reduced to its immediately previous value and the next task is considered. If t_k is schedulable, the algorithm repeats the increment and checks the worst case response time of the next potentially affected task. The time complexity of the algorithm ⑥ is $O(n^2 \cdot r)$, where n represents the number of tasks and r stands for the cost of function WCRT(π_j,γ_j) in the worst case.

5.3 Applying FPPT to AEH

In [66], it is proposed that each preemption threshold be allocated a single server, realising a static 1:N mapping model. Assume that there are 7 handlers with 3 preemption threshold levels. They therefore collectively requires 3 server threads. Servers will have static priorities, high, middle, low, and are bound to their corresponding preemtion threshold group. Consequently handlers in a preemption threshold group can only be executed by the dedicated server to the group. While it executes n handlers with p servers, where n and p represent the number of handlers and priority levels, respectively, it entails the following issues:

- Issue 1: Most of handler releasing scenarios will require the least upper bound of servers as now servers are statically allocated to their corresponding preemption thresholds. This is also a drawback of the static 1:N mapping model.
- Issue 2: It removes the effect of normal priorities as a lower regular priority handler will be executed first if it has a higher preemption threshold than a handler with a higher regular priority, but lower than the preemption threshold, which is released earlier than the lower regular priority handler. This is particularly important as this breaks the primary assumption of FPPT that is queuing released tasks according to their priorities, and executing them based on their preemption thresholds.
- Issue 3: As a result of the above, it forces a handler to produce its worst-case response time under FPPT.

The non-blocking AEH implementation [35] has been extended to solve all the issues above by taking a more dynamic approach. The extension of the non-blocking AEH implementation to support FPPT is trivial. Servers in the implementation now have to assume that each handler has 2 levels of priority, regular priority and preemption threshold. The RTSJ conveniently provides a facility to realise this, the `ImportanceParameters` class which is an additional scheduling metric that is primarily meant to be used by some priority-based scheduling algorithms during overload conditions to differentiate execution order among threads of the same priority. Note that the base priority scheduler in the RTSJ does not use the importance value in the `ImportanceParameters` class. Assuming that each non-blocking handler is given a valid priority and importance, released non-blocking handlers are queued based on their priorities and executed according to their importance, effectively scheduling them with preemption threshold. Therefore the implementation hides the use of importance from the base priority scheduler as servers will update their priorities to the importance value of the highest pending handler. As a result, servers are scheduled strictly in a priority-based manner, not an importance-based, from the scheduler's perspective. This enables RTSJ priority inversion control policies to perform correctly with servers in the non-blocking AEH implementation without any modification.

The non-blocking AEH implementation essentially uses the dynamic 1:N mapping and hence Issue 1 is not found in the implementation as it invokes a server when absolutely necessary, reducing the number of servers on average. Issue 2 and 3 are resolved by queuing released handlers based their regular priorities and executing according to their preemption threshold, preserving the primary assumption of FPPT.

5.4 Evaluation

The overall goal of applying the notion of preemption threshold to AEH in the RTSJ is to reduce the overheads of asynchronous event handling without jeopardising the schedulability of the system. The main overheads associated with schedulable objects are: context-switch time, thread control blocks (TCB) and stack size. Clearly, reducing the least upper bound of server threads will reduce the least upper bound of TCB needed. In this section, we evaluate whether FPPT reduces the number and magnitude of context-switches and stack size. This is achieved through extending the standard schedulability analysis with respect to the number of context-switches incurred and deriving an equation that calculates the least upper bound of stack usage in the worst case. Here we also present an example task set to demonstrate the benefits of applying the notion of preemption threshold to AEH.

5.4.1 The Impact of Fewer Context-Switches

The resulting preemption threshold assignment is entirely dependent on the given task set. As discussed in Section 5.2, there may be none or more valid preemption threshold assignments for a task set to be schedulable by FTTP and all the valid assignments are delimited by the minimal and maximal assignments, $\gamma_{min} \preccurlyeq \gamma \preccurlyeq \gamma_{max}$. Therefore the maximal preemption threshold assignment algorithm presented in Listing 5.1 will produce γ_{min} in the worst case and γ_{max} in the best case, for a given task set. Equation 4.6 in Section 4.5 that provides an upper bound on context-switch times for the non-blocking AEH model is presented. For the sake of convenience, the equation is copied here[1].

$$R_i = 2cs + C_i + B_i + \sum_{j \in hep(i)} \left\lceil \frac{R_i}{T_j} \right\rceil (need(j+1)2cs + C_j) \tag{5.6}$$

If the resulting preemption threshold assignment γ is in the range $\gamma_{min} \preccurlyeq \gamma \preccurlyeq (\gamma_{max} \neq \text{FPNP} = \{\forall i | \gamma_{max_i} = \pi_n\})$, the above schedulability analysis can directly be applied. If the resulting assignment γ is equal to $\gamma_{max} = \text{FPNP}$, there is only one priority level. Therefore the function *need* will always return 0, effectively removing $need(j+1)2cs$ and

[1]Note that FPPT uses the schedulability analysis based on the FPPT's own critical instant and the level-π_i busy interval, presented in Section 5.1, which differs from the above standard response time analysis based on the FPP's critical instant. However, it is the same with respect to the least upper bound of context switches by considering a preemption threshold as a regular priority, which is in the interest of this section, while the actual response times may or may not differ. The standard response time equation is still used here to help the comparison at a glance.

the schedulability analysis becomes:

$$R_i = 2cs + C_i + B_i + \sum_{j \in hep(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \tag{5.7}$$

This also depicts that only one server thread is required even in the worst case if the resulting preemption threshold assignment is $\gamma_{max} = \{\forall i | \gamma_{max_i} = \pi_n\}$. This further accelerates overall performance of the system as fewer active servers will reduce the scheduling overheads significantly – for example, the time taken to execute the clock handler can vary considerably, depending on the number of ready or blocked threads [6].

5.4.2 The Impact of Fewer Servers on Stack Size

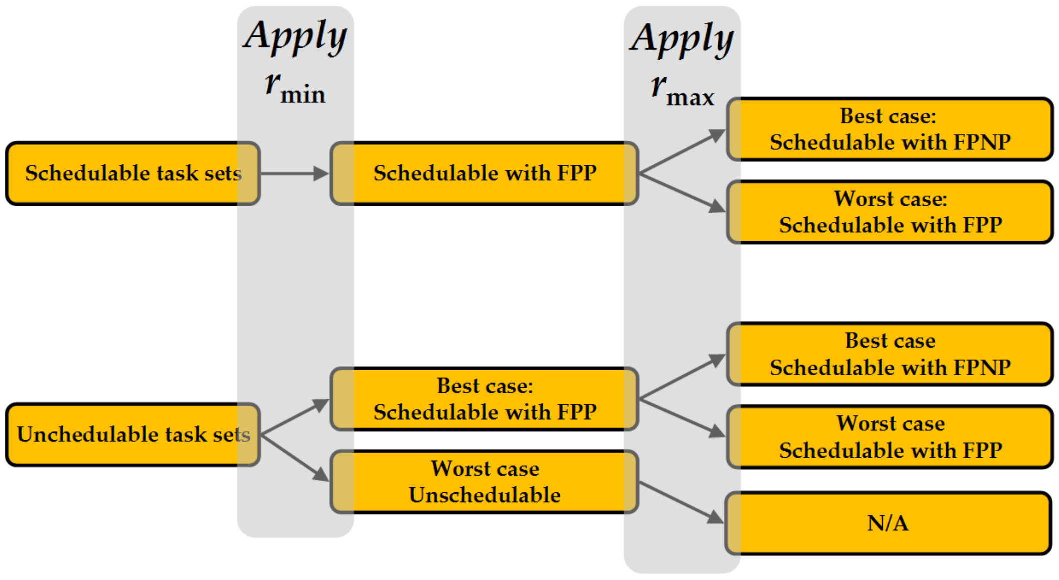
By limiting the number of active servers in the system, the amount of stack used is also be reduced. Each thread has a separate stack to store the state of its Java method invocations [63]. The state of a Java method invocation includes its local variables, the parameters with which it was invoked, its return value (if any), and intermediate calculations. The Java stack is composed of *stack frames*. A stack frame contains the state of one Java method invocation. When a thread invokes a method, the Java virtual machine pushes a new frame onto that thread's stack. When the method completes, the virtual machine pops and discards the frame for that method. For the dynamic and static 1:1 mapping model the least upper bound of stack usage in the worst case, LSU, is as the following:

$$\text{LSU} = \sum_{i=n}^{1} SU_i \tag{5.8}$$

where SU_i represents the maximum of stack usage of thread t_i. For the static 1:N mapping model, assuming that a single server thread is allocated for each priority level, LSU becomes:

$$\text{LSU} = \sum_{p=n}^{1} max_{i \in p}(SU_i) \tag{5.9}$$

where p represents a priority level from the highest to the lowest and $max_{i \in p}(SU_i)$ depicts the maximum stack size of all the tasks in the priority level p. By incorporating the notion of preemption threshold with the non-blocking AEH model, the least upper bound of stack usage can further be reduced, depending on the resulting preemption threshold assignment. If the resulting preemption threshold assignment γ is in the range $\gamma_{min} \preccurlyeq \gamma \preccurlyeq (\gamma_{max} \neq \text{FPNP} = \{\forall i | \gamma_{max_i} = \pi_n\})$, Equation 5.9, can directly be applied, which differs from Equation 5.8 for the 1:1 mapping models. If the resulting assignment γ is equal to $\gamma_{max} = \text{FPNP}$, there is only one priority level. Therefore there is only one server thread,



Apply $r_{\min}$
Apply $r_{\max}$
Schedulable task sets
Schedulable with FPP
Best case: Schedulable with FPNP
Worst case: Schedulable with FPP
Unschedulable task sets
Best case: Schedulable with FPP
Worst case Unschedulable
Best case Schedulable with FPNP
Worst case Schedulable with FPP
N/A

Figure 5.2: Categories of Task Sets w.r.t FPPT

effectively removing $\sum_{p=n}^{1}$ and the equation becomes:

$$\text{LSU} \;= max_{n\leq i\leq 1}(SU_i) \tag{5.10}$$

This depicts that the non-blocking AEH model with the notion of preemption threshold further reduces the least upper bound of stack usage of the system by reusing the stacks of fewer active server threads. If the stack used for context-switches and interrupts is considered, more improvements can be made on the least upper bound of stack usage; stacks are also used to handle context-switches and to service interrupts [13]. Fewer servers will result in fewer context-switches and interrupts will use the stack of currently active servers[2].

5.4.3 The Impact on the Number of Preemptions

In Figure 5.2, task sets are categorised based on the results of the application of FPPT. If a given task set is already schedulable by FPP, the minimal preemption threshold assignment algorithm ① will always produce the preemption threshold assignment of the task set where $\gamma_{min} = \{\forall i|\gamma_{min_i} = \pi_i\}$ and therefore the maximal preemption threshold assignment algorithm ② can directly be applied to the task set without applying γ_{min} beforehand. In such cases, $\gamma_{max} = \{\forall i|\gamma_{max_i} = \pi_i\}$ is the worst case assignment and $\gamma_{max} = \{\forall i|\gamma_{max_i} = \pi_n\}$ is the best case assignment, with respect to reducing the number

[2]This based on the assumption that interrupts do not have their own stack. This typically requires underlying hardware support.

of active priority levels. If a given task set is unschedulable by FPP, the minimal preemption threshold assignment algorithm ① will produce the preemption threshold assignment of the task set where $\gamma_{min} = \{\forall i | \gamma_{min_i} = \pi_i\}$ in the best case. In the worst case, there may be no valid preemption threshold assignment to which the maximal preemption threshold assignment algorithm ② cannot be applied as it must start with the valid preemption threshold assignment. If γ_{min} says the given task set is schedulable with FPP, γ_{max} can be applied. Consider the following task set where each task has a computation time and a deadline equal to its period (computation time, period). There are 8 tasks such that $t_8 =$ (1, 10), $t_7 =$ (1, 15), $t_6 =$ (4, 40), $t_5 =$ (10, 60), $t_4 =$ (20, 80), $t_3 =$ (15, 100), $t_2 =$ (10, 200), $t_1 =$ (16, 240), it is not schedulable by FPP as $w_1 = 293$, greater than its deadline 240. However the task set becomes schedulable by using FPPT with the preemption threshold assignment $\gamma_{min} = \{8, 7, 6, 5, 4, 4, 4, 2\}$ that is generated by the algorithm ①. The worst case response time of each task is 1, 2, 6, 18, 77, 94, 176, and 235 respectively. The preemption threshold of each task can be maximised using the algorithm ②, starting from γ_{min}, $\gamma_{max} = \{8, 8, 8, 7, 6, 6, 7, 6\}$. The worst case response time of each task becomes 11, 19, 29, 40, 60, 96, 170, and 172 respectively. Therefore FPPT is not only making the task set unschedulable by FPP schedulable, but also γ_{min} enabling the task set with 8 tasks to be feasibly scheduled with 6 servers and γ_{max} with 3 servers, even at the critical sequence. The overall stack usage is also reduced. For γ_{min}, it is the sum of the least upper bound of stack size of 6 servers while γ_{max} requires the sum of the least upper bound of stack size of 3 servers. In [65], the numbers of preemptions for two different scheduling policy, FPP and FPPT, is compared in a quantitative manner for randomly generated task sets using a uniform distribution function. The average percentage reduction in the number of preemptions for FPPT as compared to FPP is roughly between 5-31%. This also means that the same range of average percentage reduction in the number of server threads can be achieved. There is a higher percentage of the reduction in preemptions for small number of tasks, but it tapers down to less than 5% as the number of tasks increases. Also, for any given number of tasks, the number of reductions in preemptions is larger for larger values of the period range. These results are valid to the work presented in this chapter as it is based on FPPT. This does not undermine the main contribution of this chapter because FPPT is primarily introduced to schedule a task set unschedulable under FPP, and to apply FPPT to AEH in order to reduce the number of server threads required based on the preemption thresholds.

5.5 Summary

In this chapter, FPPT is discussed to reduce the number of effective priority levels for a given task set. Each task is given two priorities, a regular priority and a preemption threshold. The regular priority of a task is used to queue when released, and when a task gets the CPU its priority is raised to its preemption threshold. The task keeps the raised priority until the end of its execution at which point the priority is lowered to its regular value. Therefore FPPT allows a task to prevent preemption of tasks with higher regular priorities up to its preemption threshold, further reducing the number of effective priority levels. For any valid assignment $\gamma = \{\gamma_1, \gamma_2, ..., \gamma_n\}$, it must satisfy that $\gamma_i \in [\gamma_{max_i}, \gamma_{min_i}]$, for $1 \leq i \leq n$, which is denoted as $\gamma_{min} \preccurlyeq \gamma \preccurlyeq \gamma_{max}$. Therefore, in order to minimise the number of priority levels for a task set, the maximal assignment γ_{max} should be used as each task in a task set where the maximal assignment is applied has the highest possible preemption threshold, increasing the possibility of priority overlapping between tasks. The non-blocking AEH implementation has been extended to support FPPT to reduce the least upper bound of server threads at the critical instance which solely depends on the number of priority levels. By applying FPPT on a given task set, the least upper bound of server threads can be decreased to one, when $\gamma = \{\forall i | \gamma_i = \pi_n\}$, at best and it is the same as the existing priority levels, when $\gamma = \{\forall i | \gamma_i = \pi_i\}$, at worst. By doing so, the primary goal of AEH in the RTSJ is better achieved, enabling AEH to be truly lightweight. As a side effect of decreasing the number of servers, the number of context-switches is reduced. This saves of both space (overall stack usage) and time (context-switch).

Chapter 6

Refactoring AEH in the RTSJ

The AEH facilities in the RTSJ are criticised as lacking in configurability [38, 46, 69] as they do not provide any means for programmers to have fine control over the AEH facilities, such as the mapping between real-time server threads and handlers, and the size of the pool. As examined in Section 2.3, the current RTSJ does not provide any implementation guidelines on how asynchronous event handlers and their server threads can be implemented to guarantee the timing requirements of released handlers. Furthermore the current RTSJ implementations examined in Chapter 3 do not furnish programmers with well-defined documentation for their respective AEH approaches. As a result, programmers must rely on the underlying AEH algorithm of a RTSJ implementation for their applications and cannot project how their handlers would behave at run-time. On top of this, there are needs for the AEH subsystem to be manipulated in order to obtain a certain properties, such as efficiency. For example, if handlers with different characteristics (blocking or non-blocking, and periodic or non-periodic as presented in Chapter 4 and 5) are executed in different manners using specifically tailored AEH algorithms, more efficiency can be achieved. For these reasons, it needs the refactoring of its application programming interface (API) to give programmers more configurability. This chapter, therefore, proposes a set of AEH related classes and an interface 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 chapter shows that it allows more configurability for programmers than the current AEH API in the RTSJ does. Consequently the programmers are able to specifically tailor the AEH subsystem to fit their applications' particular needs.

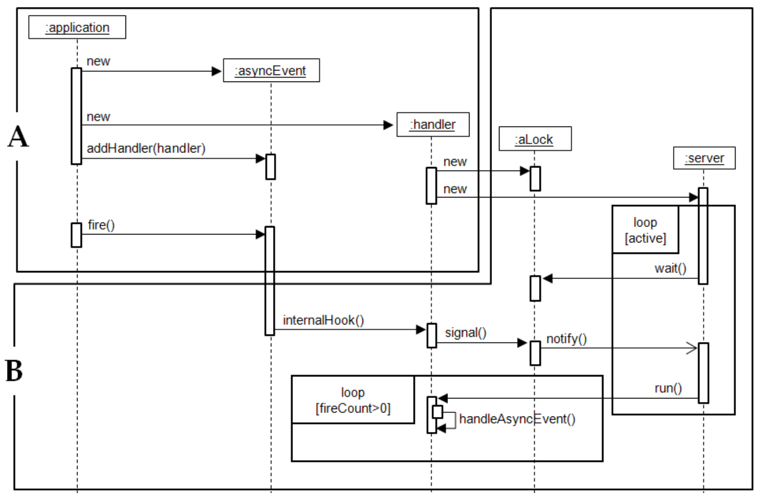


Figure 6.1: An AEH Sequence Diagram for a Simple Scenario

In Section 6.1, we give an analysis of the current AEH facilities in the RTSJ to show that only limited configurability and flexibility are provided, which prevents AEH from being adaptable for various event handling strategies. In Section 6.2 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 6.3, we present a case study of using the new AEH API framework to support an application-defined implementation of asynchronous event handlers, illustrating that a simple refactoring of the RTSJ support would allow much more flexibility and configurability towards AEH. Finally, Section 6.5 summaries this chapter.

6.1 The Configurability of AEH in the RTSJ

In order to understand fully the AEH model, it is necessary to consider how both events and event handlers are supported in the RTSJ. Figure 6.1 shows a sequence diagram for a simple application that uses `AsyncEventHandler` and `AsyncEvent` objects. Note that the sequence diagram illustrates the static 1:1 AEH mapping algorithm which is primarily used for bound asynchronous event handlers. There are two separate sections, **A** and **B**, each of which depicts the application and the implementation-specific part of the system, respectively. The application creates an 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 informs associated handlers of the occurrence of the event, is invoked. Note that the name of the internal hook method varies depending on the implementation as no requirements of the naming is specified in the RTSJ. The method may notify a server thread or may create one depending on the AEH model currently in use. Then the server thread starts to execute the released handler by calling the `handleAsyncEvent` method, which holds the application logic written by the programmer, repeatedly while the `fireCount` of the released handler is greater than 0. Therefore the `run` method of the server thread is written so that it invokes the `run` method of the handler. The RTSJ does not provide any means for programmers to have fine control over Section **B**. Therefore the section is entirely implementation-dependent and the programmer has no control over this section. Furthermore, any implementation configurability is necessarily implementation-dependent. Any program that make use of such flexibility will therefore be non-portable.

Whilst the goals of asynchronous event handling in the RTSJ are laudable, their realisation in the current version of the specification, therefore, suffer from the following limitations due to the above issue:

- A single model for all types of events handlers — all asynchronous event must be handled in the same implementation-dependent way; the `AsyncEventHandler` class provides the concrete implementation of an AEH algorithm that mandates the way of handling events and it is not possible for an application to indicate a different implementation strategy for various handlers with different characteristics in a portable way such as interrupt handlers or non-blocking handlers.

- Lack of implementation configurability — the `AsyncEventHandler` class provides no facilities for the programmer to configure its AEH and therefore the application is unable to finely tune its AEH components such as the thread pool or the AEH mapping algorithm [35].

The above problems in the RTSJ severely limit the configurability and the flexibility of the AEH implementation and consequently the RTSJ's model has been criticised as lacking in this area [35, 46, 69]. In the following section, a refactoring of the RTSJ support is presented to allow the above limitations to be removed.

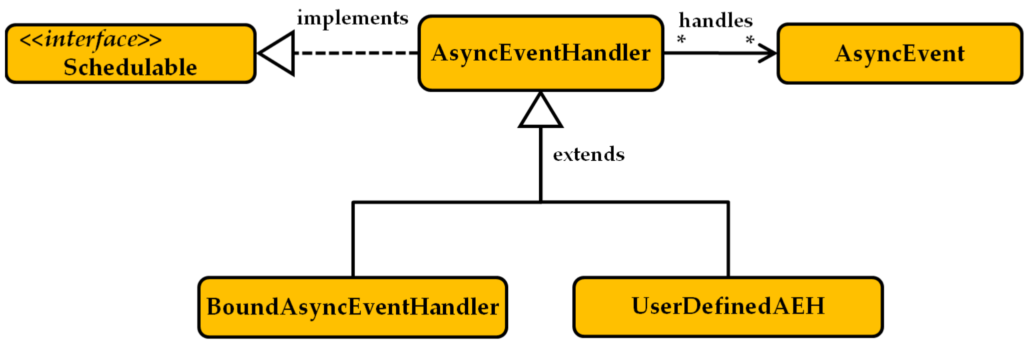


Figure 6.2: The AEH API in the RTSJ

6.2 Refactoring the AEH API in the RTSJ

As discussed, the RTSJ does not provide facilities that enable the application to configure its AEH components and to regulate the run-time behaviour of its server threads. Also the RTSJ implementations presented in Chapter 3 neither provides adequate documentation for their respective run-time behaviour of AEH nor furnishes it with a certain level of configurability. In Chapter 2 the overall design of the RTS AEH facilities were discussed. The core design of the AEH API is presented in Figure 6.2 and is based on two premises:

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

In this section we remove these premises and refactor the current AEH 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 6.3. 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 to 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,

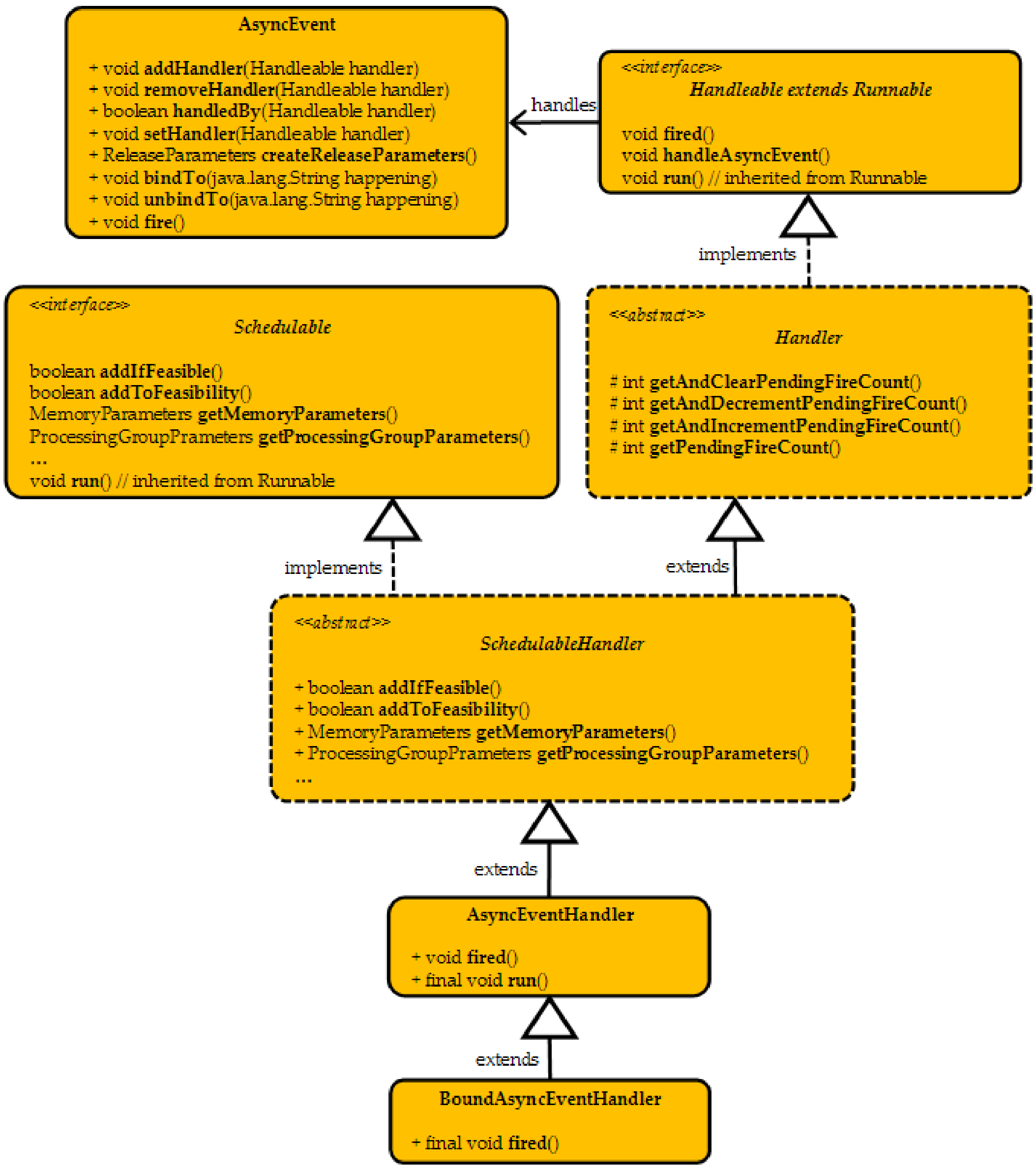

AsyncEvent
+ void **addHandler**(Handleable handler)
+ void **removeHandler**(Handleable handler)
+ boolean **handledBy**(Handleable handler)
+ void **setHandler**(Handleable handler)
+ ReleaseParameters **createReleaseParameters**()
+ void **bindTo**(java.lang.String happening)
+ void **unbindTo**(java.lang.String happening)
+ void **fire**()
handles
<<interface>>
Handleable extends Runnable
void **fired**()
void **handleAsyncEvent**()
void **run**() // inherited from Runnable
implements
<<interface>>
Schedulable
boolean **addIfFeasible**()
boolean **addToFeasibility**()
MemoryParameters **getMemoryParameters**()
ProcessingGroupPrameters **getProcessingGroupParameters**()
...
void **run**() // inherited from Runnable
<<abstract>>
Handler
# int **getAndClearPendingFireCount**()
# int **getAndDecrementPendingFireCount**()
# int **getAndIncrementPendingFireCount**()
# int **getPendingFireCount**()
implements
extends
<<abstract>>
SchedulableHandler
+ boolean **addIfFeasible**()
+ boolean **addToFeasibility**()
+ MemoryParameters **getMemoryParameters**()
+ ProcessingGroupPrameters **getProcessingGroupParameters**()
...
extends
AsyncEventHandler
+ void **fired**()
+ final void **run**()
extends
BoundAsyncEventHandler
+ final void **fired**()

Figure 6.3: The Refactored Configurable AEH for the RTSJ

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 the `Runnable` interface and it 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. The `BoundAsyncEventHandler` class extends the `AsyncEventHandler` class as defined in the current RTSJ. The `fired` method in the `BoundAsyncEventHandler` class is declared as final to shield its internal algorithm from being modified.

In the RTSJ, the interaction between the `AsyncEventHandler` class and the `AsyncEvent` class is not clearly defined and hence is implementation-specific in an inconsistent way across different implementations. The RTSJ merely states that [5]:

> 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 phrase 'released for execution' is not clear enough, which possibly results in different interpretation. 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). The server thread in OVM then calls the `run` method of the handler, which in turn invokes the `handleAsyncEvent` method repeatedly while `fireCount` is greater than zero. In 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. Therefore, here the interaction between the `Handleable` interface (or the `AsyncEventHandler` class) and the `AsyncEvent` is explicitly defined to precisely describe their corrective behaviour, such that:

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

- 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:

The proposed API structure and the formal definition of the relationship between the `Handleable` interface and the `AsyncEvent` class allows the application to indicate a different hierarchical implementation strategy for various handlers with different characteristics. This can be achieved by extending the refactored AEH API in several ways that fit the application's particular needs. 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 6.4. 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 [69]:

  1. handlers in separate servers can be organised so that they do not need to be synchronised,
  2. handlers with tight deadlines can be kept separate from handlers with long execution time,

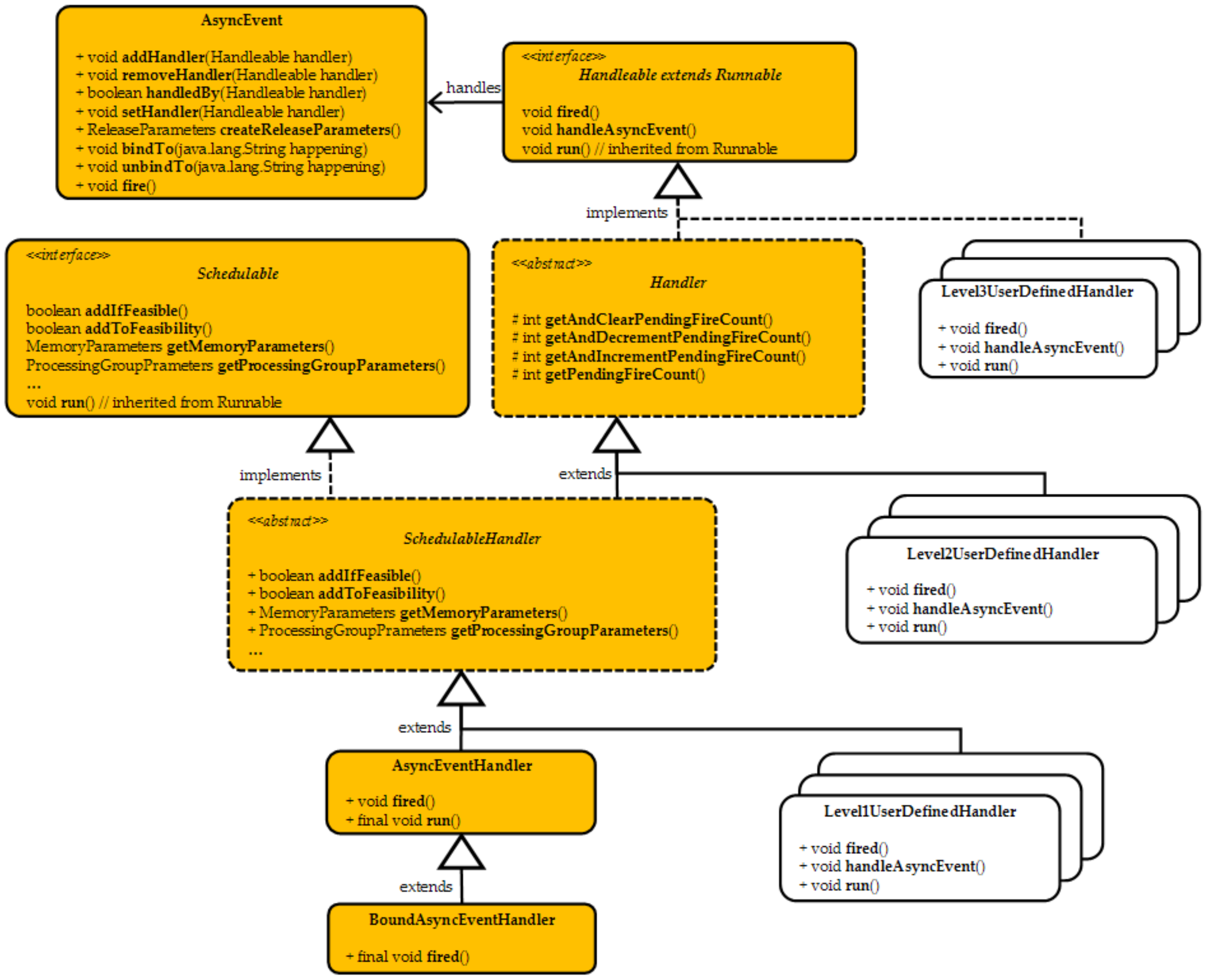

AsyncEvent
+ void addHandler(Handleable handler)
+ void removeHandler(Handleable handler)
+ boolean handledBy(Handleable handler)
+ void setHandler(Handleable handler)
+ ReleaseParameters createReleaseParameters()
+ void bindTo(java.lang.String happening)
+ void unbindTo(java.lang.String happening)
+ void fire()
handles
<<interface>>
Handleable extends Runnable
void fired()
void handleAsyncEvent()
void run() // inherited from Runnable
implements
<<interface>>
Schedulable
boolean addIfFeasible()
boolean addToFeasibility()
MemoryParameters getMemoryParameters()
ProcessingGroupPrameters getProcessingGroupParameters()
...
void run() // inherited from Runnable
<<abstract>>
Handler
# int getAndClearPendingFireCount()
# int getAndDecrementPendingFireCount()
# int getAndIncrementPendingFireCount()
# int getPendingFireCount()
Level3UserDefinedHandler
+ void fired()
+ void handleAsyncEvent()
+ void run()
implements
extends
<<abstract>>
SchedulableHandler
+ boolean addIfFeasible()
+ boolean addToFeasibility()
+ MemoryParameters getMemoryParameters()
+ ProcessingGroupPrameters getProcessingGroupParameters()
...
Level2UserDefinedHandler
+ void fired()
+ void handleAsyncEvent()
+ void run()
extends
AsyncEventHandler
+ void fired()
+ final void run()
Level1UserDefinedHandler
+ void fired()
+ void handleAsyncEvent()
+ void run()
extends
BoundAsyncEventHandler
+ final void fired()

Figure 6.4: Extended Application Classes using the Refactored Configurable AEH for the RTSJ

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 that are presented in Section 4.3.

- It can provide non-schedulable handlers by extending the `Handler` class as illustrated as the `Level2UserDefinedHandler` class in Figure 6.4. 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 [17].
- It can provide its own version of the `Handler` class should the need arise. The `Level3UserDefinedHandler` class in Figure 6.4 illustrates this possibility. `Handle-ables` at this level can similarly be implemented as `Leve3UserDefinedHandler`. Here, however, the notion of `fireCount` can also be bypassed.
- Instances of the class that extends the `AsyncEventHandler` class and overrides the `handleAsyncEvent` 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`.

6.3 Using the Refactored AEH API

We have implemented the refactored configurable AEH in jRate running on top of an open-source RTOS, MarTE OS [53], 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 6.5 shows the simplest AEH model, that extends the `SchedulableHandler` class and is therefore a `Level1UserDefinedHandler`. The AEH model is used in the RI [60] and creates a server every time a handler is released. All the activities done in the sequence diagram now belong with the application-specific part of the system unlike the previous sequence diagram in Figure 6.1, that uses the current AEH API of the RTSJ. The `fired`

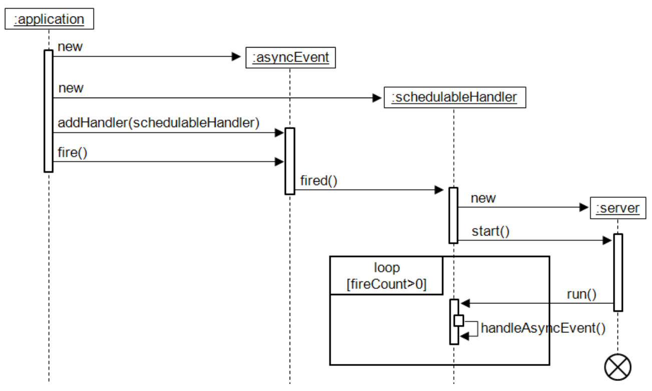


Figure 6.5: Dynamic-server-creation AEH Model

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. The efficient AEH models presented in Chapter 4 and 5 also have been implemented on top of the refactored AEH API as a `Level1UserDefinedHandler` class. The porting is trivial as the only required modification is for them to extend the `SchedulableHandler` class and change the name of the internal hook method to `fired`.

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.

6.4 Evaluation

The refactored AEH API presented in Section 6.2 newly introduces an interface and two abstract classes to the current AEH API in the RTSJ. This is in order to give handlers different characteristics and allow the application to have full control over its AEH components, including the AEH mapping algorithm which governs the run-time behaviour of handlers. In this section, we discuss relevant issues when refactoring the current AEH API to the form of the AEH API presented in Section 6.2. The issues are backward compatibility, safety, and API overhead. They are discussed in terms of the Java Language Specification [48] and then are manually tested to confirm their resulting behaviour.

6.4.1 Backward Compatibility

For source compatibility, the source code must be compiled before running. For the binary compatibility, on the other hand, no recompilation of the source code is needed. The refactored AEH API preserves backward source compatibility with applications written in accordance with the current RTSJ. This can be easily validated by the following facts:

- Instances of the current `AsyncEventHandler` class still remain as instances of the

`AsyncEventHandler` class in the refactored AEH API.

- The current AEH API in the RTSJ hides the methods, which govern the run-time behaviour of handlers, from the application. As a consequence, all the possible methods to be overridden or used by the application, or to be inherited from the current `AsyncEventHadler` class remain the same and are also available in the refactored AEH API.

However, binary compatibility is essentially not supported due to changes made to the `AsyncEvent` class. According to the Java Language Specification [48], 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 existing addHandler() method in the current RTSJ
// that calls the new addHandler(Handleable handler) method
// after typecasting the parameter handler to handleable
public void addHandler(AsyncEventHandler handler) {
   if(handler instanceof Handleable) {
      Handleable handleable = handler;
         addHandler(handleable);
      }
   }
}
```

Therefore the `AsyncEvent` class will have two `addHandler` method with different signatures. The application written in accordance with the current API will be linked to the `addHandler(AsyncEventHandler handler)` method, which in turn calls the new `addHandler(Handleable handler)` method after typecasting the passed-in parameter, `handler`, to a `handleable`. Note that, handlers in the refactored AEH API will always

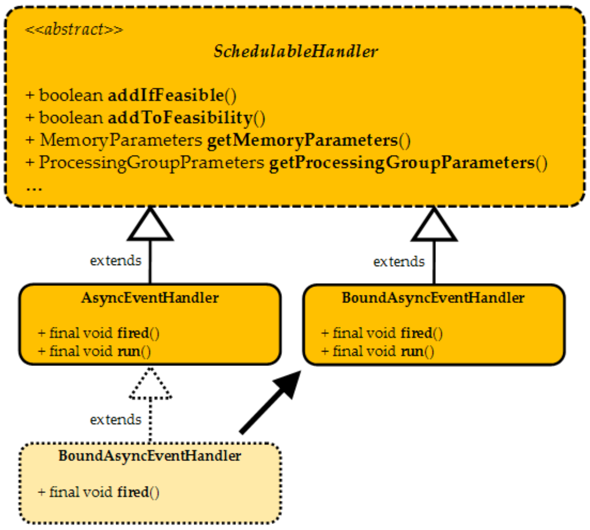


Figure 6.6: Encapsulation of the AEH Algorithm

be an instance of `Handleable`. By doing this, therefore, the refactored AEH API can also preserve backward binary compatibility.

6.4.2 Safety

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` class 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. Nevertheless, the `fired` method in the `AsyncEventHandler` class can be overridden by the programmer. Programmers must not do so if it is intended to use the default AEH algorithm provided by the underlying AEH implementation. One solution for shielding the AEH algorithm from being modified in the refactored AEH API is to move the `BoundAsyncEventHandler` class to extend the `SchedulableHandler` class directly, not via the `AsyncEventHandler` class, as showin in Figure 6.6. This enables the `fired` method in the two subclasses of the `SchedulableHandler` class to be declared as final, preventing the method from being accidentally altered. Note that, as shown in Figure 6.6, the `run` method needs to be declared in the `BoundAsyncEventHandler` class as well. Either way, the `AsyncEventHandler` and

`BoundAsyncEventHandler` classes can provide the default implementation of its AEH as the current RTSJ specifies.

If the `BoundAsyncEventHandler` class extends the `SchedulableHandler` class directly, not via the `AsyncEventHandler` class, to declare the `fired` method as final shielding the AEH algorithm from being altered, it, however, may affect both source and binary backward compatibility. This is because the `BoundAsyncEventHandler` class is no longer an instance of the `AsyncEventHandler` class. According to the Java Language Specification [48], if a change to the direct superclass or the set of direct superinterfaces results in any class or interface no longer being a superclass or superinterface, respectively, then link-time errors may result if pre-existing binaries are loaded with the binary of the modified class. Such changes are not recommended for widely distributed classes. Consequently, there is a trade-off, depending on the position of the `BoundAsyncEventHandler` class.

6.4.3 API Overhead

Although the refactored AEH API introduces an interface and two abstract classes additionally to the current AEH API in the RTSJ, there are no extra overheads imposed by the use of the refactored AEH API. This is due to the fact that there are no additional constructors, methods, or variables which are newly defined in the refactored AEH API and are not found in the current AEH API. Essentially, all the constructors, methods, and instance variables defined in the `AsyncEventHandler` class of the current AEH API are moved and put to the `Handleable` interface, and the `Handler` and `SchedulableHandler` classes, accordingly. The `fired` method is the only one that newly added to the refactored AEH API. However, the method is declared in the `Handleable` interface and hence does not provide its implementation (i.e. without a method body). Furthermore, strictly speaking, the method already exists in the current AEH API with different names as a internal structure and is simply made open to the programmer to provide an application-specific implementation in the refactored AEH API. Therefore there is no added overhead imposed by the use of the refactored AEH API compared to the current one. This is validated based on the following facts derived from the Java Language Specification [48]:

- Whenever a new class instance is created, memory space is allocated for it with room for all the instance variables declared in the class type and all the instance variables declared in each superclass of the class type, including all the instance variables that may be hidden. Unless there is sufficient space available to allocate memory for the object, all the instance variables in the new object, including those declared in superclasses, are initialised to their default values. As a consequence, when a class instance is created, instances of its superclasses are only initialised[1], not created. Therefore, for example, instances of the `AsyncEventHandler` class in the refactored AEH API have the same overhead as those of the `AsyncEventHandler` class in the current AEH API do, in terms of memory, as there are no additional instance or static variables and methods in the refactored AEH API.

- Before a reference to the newly created object is returned as the result of a class instance creation, associated constructors (i.e. its own constructor and its superclass's constructor) are processed to initialise the new object. As the refactored AEH API does not define any additional constructor, the only overhead in terms of time is to implicitly invoke the two more default constructors with empty body, which are provided automatically by the RT-JVM compiler as there are two super classes, `Handler` and `SchedulableHandler`. Note that before a class is initialised, its direct superclass must be initialised, but interfaces implemented by the class need not be initialised. Similarly, the superinterfaces of an interface need not be initialised before the interface is initialised. Therefore the time overhead imposed by the use of the refactored AEH API is trivial and hence negligible. This fact is borne out by repeating the experiment in Chapter 4.4.

6.5 Summary

This chapter has considered the rationale for refactoring asynchronous event handling in the Real-Time Specification for Java. As the RTSJ does not provide any configurable facilities for AEH, all the implementations examined in this thesis neither furnish programmers with well-defined documentation 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. As an usage example of the refactored AEH API, this chapter presented a simple AEH implementation model that uses the dynamic 1:1 mapping model with run-time server creation. The refactored AEH API lifts the limitations of the current

[1]In Java, initialisation of a class consists of executing its static initialisers and the initialisers for static fields declared in the class.

AEH version of the RTSJ, by giving programmers 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 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 [35] 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 behaviour 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 thesis, 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.

- No additional API overhead – the refactored AEH API does not have additional API overhead in terms of both time and memory than the current AEH API in the RTSJ do.

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 its particular needs.

Chapter 7

Conclusions and Future Work

The aim of this chapter is to summarise the work of this thesis and its contributions, as well as to provide insight into further work that could be pursued. The primary goals of the thesis are to augment the current AEH in the RTSJ in order for it to be more efficient and more flexible so that its primary goal, the lightweightness, can be better achieved. In particular, two major aspects of work have been proposed as follows.

- Providing a well-defined guideline to achieve the primary goal of AEH in the RTSJ, the lightweightness, and
- Facilitating comprehensive configurability over AEH components so that the AEH subsystem can specifically be tailored to fit the application's particular needs.

The following subsections give a summary of the major contributions of the thesis and directions for further work in the light of this thesis. Note that, evaluation has been performed on a per chapter basis.

7.1 Summary of Contributions

A summary of the achievements obtained by the described research work and major contributions of this thesis are given below. These achievements are assessed based on the four objectives specified in Chapter 1 as follows:
The current version of the RTSJ does not facilitate either a lightweight concurrency mechanism or comprehensive configurability over its AEH. The efficiency and flexibility of AEH in the RTSJ can be better achieved if the implementation

Proposition I: is aware of whether asynchronous event handlers are

- Blocking or Non-blocking, and
- Periodic (Hard) or Non-periodic (Soft).

to be able to reduce overhead both in time and space, and still meet the application's timing requirements.

Proposition II: facilitates comprehensive configurability over its AEH facilities as the lack of the property inhibits the scalability of the RTSJ implementation from scaling to systems of hundreds of events. The problem is exacerbated by the presence of asynchronous event handlers with different characteristics presented in Proposition I.

The demonstration of this research propositions follows from delivering the five major objectives set out in Chapter 1 of the thesis as shown below:

Objective 1 Investigating and implementing AEH mapping algorithms that use the minimum number of server threads,

Objective 2 Drawing the distinction between blocking and non-blocking handlers to further reduce the required number of server threads,

Objective 3 Separating non-aperiodic and aperiodic handlers thereby allowing the use of preemption threshold technique for hard (or periodic) handlers: this technique lets the AEH subsystem further reduces the number of server threads for execution of released handlers even in the worst case.

Objective 4 Refactoring of the current AEH API in the RTSJ, which results in giving programmers full configurability over AEH subsystem,

In Chapter 2, asynchronous event handling facilities in high-level programming languages, middleware, and distributed systems are investigated and examined in terms of helping the application's predictable and efficient behaviour in the context of the concurrent programming models, thread-based and event-based. Either directly or indirectly they support a notion of threads and a level of asynchrony to facilitate concurrent programming. Regardless of their internal and external structures they can be classified into two classes with respect to the mapping between the executionees and the executioners. One is the single server approach and the other is the dynamic or static 1:1 mapping. Some of the investigated languages or middleware systems are able to support the both form. However there is no framework that provides asynchronous event handling with a fewer number of server threads than the number of handlers concurrently active. Asynchronous event handling in the RTSJ was discussed in terms of its basic API model and implementation. The RTSJ allows considerable freedom in the implementation of asynchronous event handling. We have also considered various models for asynchronous event handling from the single server approach to the simple thread per handler model, found in the literature. The chapter is concluded by identifying the outstanding issues associated with AEH in the RTSJ, efficiency and flexibility.

In Chapter 3, we examined various asynchronous event handling algorithms in some popular RTSJ implementations with respect to efficiency and flexibility. The followings summarise their respective approach:

- RI - Dynamic 1:1 mapping
- OVM - Static 1:1 mapping
- Jamaica - Static 1:N mapping
- jRate - Dynamic 1:1 mapping using the Leader/Followers design pattern
- Java RTS - Dynamic 1:N mapping using a thread pool

The only AEH implementation which offers both synchronisation and space decoupling is provided in Java RTS by using a thread pool. Java RTS AEH is also the only implementation that tries to reduce the overall number of server threads required on average without causing priority-inversion. To evaluate asynchronous event handling implementation of the two RTSJ implementations, jRate and Java RTS, UPPAAL has been extensively used. The two AEH automata models for each implementation have been designed, simulated and verified. They have shown to be consistent with the RTSJ and the properties including reachability, safety and liveness have been verified successfully. The verifier in UPPAAL has generated some measurements for resource usage and overheads using the AEH automata models. The verification results show that the jRate AEH model uses the same number of server threads as the number of simultaneously released handlers increase regardless of the sequence of event firings. The Java RTS AEH model, on the other hand, require a smaller number of server threads on average although the same number of server threads are required in the worst case.

The results from the simulation and the verification demonstrates that the AEH model used in Java RTS is relatively efficient by allowing server threads to execute multiple handlers under certain event-firing sequences. However, there are occasions when the Java

RTS AEH model constantly causes unnecessary server-switching, multiple-server switching phenomenon (MSSP), incurring more context-switches on average. The MSSP was classified as a concurrency-related issue (such as priority inversion) and takes place whenever the current running server changes its priority to that of the handler, which it will next execute, due to queue replacement policy in that most real-time OSs and middleware 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 Java RTS AEH implementation in order for the current running server to reflect the priority of the handler that it currently executes.

In Chapter 4, more efficient models for asynchronous event handling were proposed and implemented. The underlying idea beneath the proposed models is to use as few servers as possible. This is to achieve the primary intention of AEH in the RTSJ. they are designed to serve handlers with different characteristics, blocking and non-blocking. To validate the idea, the dynamic 1:N mapping model was discussed and emhpasised for its efficiency and scalability. Based on the dynamic 1:N mapping model, the notion of critical sequences for blocking and non-blocking handlers is defined to state the condition under which the least upper bound of server threads is required. Each blocking handler requires a single server and non-blocking handlers collectively requires the number of servers as the number of priority levels. Based on this, the equations that calculate the least upper bound of servers required for both blocking and non-blocking handlers were derived for a given set of handlers. The blocking and non-blocking AEH implementations are constructed, based upon the dynamic 1:N mapping model and utilise the critical sequences.

The blocking AEH model allocates a single server to each blocking handler and the non-blocking AEH implementation only invokes a server to cater for preemption. The 1:1 mapping model, such as bound asynchronous event handlers in the current RTSJ, always requires the least upper bound of servers unless handlers are released in a discrete manner. On the other hand, the blocking AEH model will require the least upper bound of servers if and only if a critical sequence occurs. The non-blocking AEH model further reduces the least upper bound of servers by taking advantage of non-blocking handlers. It requires the same number of servers as that of the priority levels in the system, even in the critical sequence. All the other handler releasing scenarios for the non-blocking AEH model require fewer servers than the least upper bound of servers in the critical sequence. The proposed models have also demonstrated its capability to avoid the MSSP. The performance test results indicate that current AEH implementations of the RTSJ could be optimised to use as few servers as possible, incurring less overhead. The blocking and non-blocking AEH models are prime examples of such an optimised model and on average requires the smallest number of servers and context-switches, based on their corresponding critical sequences. Also the AEH algorithms used in the blocking and non-blocking implementations are language-independent. This is because the algorithms do not use Java-specific facilities. They can be applied to any programming languages that supports the notion of thread and basic thread management facilities such as priority change and mutex. However it must be noted that, the queuing policy of the priority change of a thread depends on the underlying operating system and hence it is critical to understand the exact behaviour of priority change to properly adapt the AEH algorithms to another programming language, built on top of a different OS. Therefore, the chapters from 2 to 4, collectively achieve Objective 1 and 2. The explicit contributions of these chapters are as follows:

- A better understanding of the facilities that affect the mapping between event handlers and server threads.
- Identification of a new real-time concurrency-related problem, the MSSP.
- An evaluation of the number of server threads required based on the releasing sequence and handler's characteristics.
- A definition of critical sequences for blocking and non-blocking handlers.
- Two new asynchronous event handling algorithms that utilise the definition of critical sequences and also avoids the MSSP.

In Chapter 5, fixed-priority preemptive scheduling with preemption threshold (FPPT) is discussed to reduce the number of effective priority levels for a given task set. Each task is given two priorities, a regular priority and a preemption threshold. The regular priority of a task is used to queue when released, and when a task gets the CPU its priority is raised to its preemption threshold. The task keeps the raised priority until the end of its execution at which point the priority is lowered to its regular value. Therefore FPPT allows a task to prevent preemption of tasks with higher regular priorities up to its preemption threshold, further reducing the number of effective priority levels. For any valid assignment $\gamma = \{\gamma_1, \gamma_2, ..., \gamma_n\}$, it must satisfy that $\gamma_i \in [\gamma_{max_i}, \gamma_{min_i}]$, for $1 \leq i \leq n$, which is denoted as $\gamma_{min} \preccurlyeq \gamma \preccurlyeq \gamma_{max}$. Therefore, in order to minimise the number of effective priority levels for a task set, the maximal assignment γ_{max} should be used as each task in a task set where the maximal assignment is applied has the highest possible preemption threshold, increasing the possibility of priority overlapping between tasks. The non-blocking AEH implementation has been extended to support FPPT to reduce the least upper bound of servers required at the critical instance which utterly depends on the number of priority levels. Therefore the objective 3 is delivered by better achieving the primary goal of AEH in the RTSJ, and therefore enabling AEH to be truly lightweight. Note that by the use of server techniques for soft (or aperiodic) handlers via the notion of `ProcessingGroupParameters`, the impact of such handlers on the overall schedulability of the application can be bound [71]. The explicit contribution is the application of FPPT to a new area of asynchronous event handling, which enables further reduction of the number of server threads, by limiting the number of effective priority levels in the system without jeopardising the current schedulability.

In Chapter 6 we presented the refactored AEH API for the RTSJ. As the RTSJ does not provide any configurable facilities for AEH, Most of the implementations examined in this thesis neither furnish programmers with well-defined documentation 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, the configurable AEH API for the RTSJ was 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 refactored AEH API lifts the limitations of the current AEH version of the RTSJ, by giving programmers comprehensive configurability. Therefore Objective 4 is delivered by introducing the new AEH API which can be specifically tailored to fit the application's particular needs. The explicit contribution is the development of a new framework for configurable AEH in the RTSJ.

Accomplishing these aims can offer a more efficient and flexible way to constructing AEH subsystem in the real-time Java environment. These achievements can facilitate the development of scalable applications that can specifically be tailored to fit the application's particular needs. This further enables the primary goal of AEH in the RTSJ to be achieved by providing the lightweight concurrency mechanism.

7.2 Possible Directions for Future Research

The results presented in this thesis can be used as building blocks for constructing a thread pool framework that can be used with AEH in the RTSJ. A number of areas in which further work could be conducted to construct an efficient and configurable thread pool framework are suggested in the section.

The way of adopting FPPT to AEH in the RTSJ, presented in Chapter 5, is to shield the scheduler from the extra information (i.e. preemption threshold of a handler), in order for existing mechanisms, such as priority inversion control and priority-based scheduling, to work correctly without modifying the behaviour of the scheduler. However this approach prevents the use of FPPT with normal real-time threads in the RTSJ. This is mainly because server threads will be executed in competition with real-time threads that do not use the notion of preemption threshold. As a result, real-time threads cannot preempt server threads with a lower regular priority, but with higher or equal preemption threshold. This will undermine the existing schedulability analysis of the non-server real-time threads in the system. Therefore the non-blocking AEH implementation can only be used with systems that are composed of only non-blocking asynchronous event handlers. Blocking handlers can also be used based on the assumption that the blocking AEH implementation also supports the notion of preemption threshold. In order to use the notion with normal real-time threads, it is imperative to modify the scheduler so that it is aware of preemption threshold to schedule real-time threads, both server and non-server, according to their importance parameters. This is due to the fact that the base scheduler (i.e. `PriorityScheduler`) in the RTSJ does not use the importance value in the `ImportanceParameters` subclass of `PriorityParameters`. If such a scheduler is provided, it is possible to schedule server threads using FPPT without affecting the schedulability of non-server real-time threads in the system.

The priority assignment algorithms for FPPT presented in Chapter 5 allow the overlapping of the preemption thresholds. This means that a lower priority task may have a higher preemption threshold than that of a higher priority task. This fact also contributes to Issue 2 and 3 in Section 5.3 for the static 1:N mapping. If the overlapping possibility is removed, it is possible to assign a single server thread per priority level without having the associated issues, while preserving the advantages of the static 1:N mapping model such as ease of implementation. In order to achieve this, a new preemption threshold assignment, that does not allow the overlapping, should be derived.

Currently, semantics for the base scheduler essentially assume a uniprocessor execution environment. While implementations of the RTSJ are not precluded from supporting multiprocessor execution environments, no explicit consideration for such environments has been given in the specification [5]. As multiprocessor systems, particularly symmetric multiprocessor (SMP), are becoming prevalent, addressing issues related to such systems to be used with AEH in the RTSJ would be timely. As the current RTSJ provides no mechanism that would allow the programmer to tie happenings and their handlers to particular processors, it is not clear for the implementer to map the released handler and the available processors. In [68], a possible direction for the RTSJ towards multiprocessor systems is proposed, via supporting the notion of CPU affinity for schedulable objects. Its dispatching model assumes each schedulable object has its own execution engine. This, however, no longer holds for 1:N mapping models proposed in this thesis, where one or more handlers share an execution engine (i.e. server thread). Therefore the extensibility of the proposed AEH models in the thesis should be validated in order to be used in multiprocessor systems.

7.3 Final Words

In this thesis, we have investigated how to provide an efficient and flexible framework for asynchronous event handling in the real-time specification for Java.

In order to strengthen the efficiency of AEH, the number of server threads must be minimised. This inevitably requires an adequate mapping model that maps server threads to released handlers with efficiency. In the literature, it is however unclear how to achieve this efficiency. Most of real-time systems that require this mapping between executioners and executionees use a simple model (e.g. a dynamic 1:1 mapping, such as the Leader/-Followers design pattern). This does not achieve the efficiency as the same number of server threads will be required as the number of simultaneously released handlers. We first defined the notion of critical sequences for blocking and non-blocking event handlers. The blocking and non-blocking AEH models presented in this thesis utilise the notion to achieve the efficiency by using as few server threads as possible, depending on the releasing sequence of handlers. We also proved that the same number of server threads is required as the number of priority levels in the system for non-blocking handlers even in the worst-case. To further accelerate the efficiency using this fact, the fixed-priority preemptive scheduling with preemption threshold is applied to AEH in the RTSJ. This ultimately enables the AEH subsystem to use a smaller number of server threads (than that of existing priority levels) for executing a larger number of handlers even in the worst-case.

The current RTSJ does not provide any configurable facilities in order for programmers to finely tune their applications with respect to AEH. This severely limits the flexibility of AEH in the RTSJ. We refactored the current AEH API to provide programmers with comprehensive configurability over their AEH models. The refactored AEH API is shown that it is backward-compatible and its additional overhead compared to the current AEH API is negligible. It offers many advantages for programmers such as multiple models for various handlers with different characteristics, implementation configurability and a variety of views of the notion of sporadic handlers. As a result, the refactored AEH API provides significant benefits in terms of application's extensibility and scalability, which can be achieved by specifically tailoring the application to fit its particular needs.

The results obtained in this thesis are therefore sufficient to make us believe that the first step is steadily taken towards providing an efficient and flexible framework for asynchronous event handling in the real-time specification for Java.

Bibliography

[1] Jonathan S. Anderson and E. Douglas Jensen. Distributed Real-Time Specification for Java: A Status Report. In *JTRES '06: Proceedings of the 4th International Workshop on Java Technologies for Real-Time and Embedded Systems*, New York, NY, USA, 2006. ACM.

[2] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. J. Wellings. Applying New Scheduling Theory to Static Priority Pre-Emptive Scheduling. *Software Engineering Journal*, 8:284–292, 1993.

[3] Pete Becker. Working Draft, Standard for Programming Language C++, 2009.

[4] G. Behrmann, A. David, and Kim G. Larsen. A Tutorial on Uppaal. *Lecture Notes in Computer Science*, 3185:200–236, 2004.

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

[6] Alan Burns and Andy Wellings. *Real-Time Systems and Programming Languages Third Edition.* Addison Wesley, UK, 2001.

[7] Alan Burns and Andy Wellings. *Concurrent and Real-Time Programming in Ada.* Cambridge University Press, USA, 2007.

[8] Georgio Buttazzo. *Hard Real-Time Computing Systems.* Springer, USA, 2004.

[9] Jiongxiong Chen. *Extensions to Fixed Priority with Preemption Threshold and Reservation-Based Scheduling.* PhD thesis, Waterloo University, Waterloo, Ontario, Canada, 2005.

[10] Microsoft Corporation. C# Language Specification Version 3.0, 2007.

[11] Microsoft Corporation. .NET Framework Developer Center: .NET Framework 3.5, 2007.

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

[13] R.I. Davis, N. Merriam, and N.J. Tracey. How Embedded Applications Using an RTOS Can Stay within On-chip Memory Limits. In *12th EuroMicro Conference on Real-Time Systems*, pages 71–77, Los Alamitos, CA, USA, 2000. IEEE Computer Society.

[14] Kurt Debattista, Kevin Vella, and Joseph Cordina. Cache-Affinity Scheduling for Fine Grain Multithreading. In *Proceedings of Communicating Process Architectures 2002*, pages 135–146, Nieuwe Hemweg 6B, 1013 BG Amsterdam, The Netherlands, 2002. IOS Press.

[15] A. Diaz, I. Ripoll., and A. Crespo. On Integrating POSIX Signals into a Real-Time Operating System. In *7th Real-Time Linux Workshop*, 2005.

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

[17] Peter Dibble. and Andy 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.

[18] Ludwig D.J. Eggermont. Embedded Systems Roadmap 2002: Vision on technology for the future of PROGRESS, 2002.

[19] P. Th. Eugster, P. A. Felber, R. Guerraoui, and A. m. Kermarrec. The Many Faces of Publish/Subscribe. *ACM Computing Surveys*, 35:114–131, 2003.

[20] Linux Foundation. Linux Standard Base (LSB) Specifications, 2008. http://www.linuxfoundation.org/en/Specifications.

[21] Dabek Frank, Zeldovich Nickolai, Kaashoek Frans, Mazières David, and Morris Robert. Event-driven Programming for Robust Software. In *Proceedings of the 10th workshop on ACM SIGOPS European workshop*, pages 186–189, New York, NY, USA, 2002. ACM.

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

[23] DRTSJ Expert Group. Java Specification Request 50: Distributed Real-Time Specification, 2000.

[24] Object Management Group. Real-Time CORBA Specification 2.0: Dynamic Scheduling, 2003. http://www.omg.org/docs/formal/03-11-01.pdf.

[25] Object Management Group. CORBA Services: Event Service Specification 2nd Edition, 2004. http://www.omg.org/docs/formal/04-10-02.pdf.

[26] Object Management Group. Notification Service Version 1.1, 2004. http://www.omg.org/technology/documents/formal/notification_service.htm.

[27] Object Management Group. Data Distribution Service Specification for Real-Time Systems Version 1.2, 2007. http://www.omg.org/technology/documents/formal/data_distribution.htm.

[28] Timothy H. Harrison, David L. Levine, and Douglas C. Schmidt. The Design and Performance of a Real-time CORBA Event Service. In *In Proceedings of OOPSLA '97*, pages 184–199. ACM, 1997.

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

[30] Lauer Hugh and Needham Roger. On the Duality of Operating System Structures. *SIGOPS Oper. Syst. Rev.*, 13(2):3–19, 1979.

[31] IEEE. Potable Operating System Interface: IEEE Std 1003.1b (Real-time extensions), IEEE Std 1003.1c (Threads extensions), 1995. http://standards.ieee.org/regauth/posix/.

[32] Lina Khatib, Nicola Muscettola, and Klaus Havelund. Verification of Plan Models Using UPPAAL. *Lecture Notes in Computer Science*, 1871:114–122, 2001.

[33] MinSeong Kim. Asynchronous Event Handling. Technical report, The University of York, 2006.

[34] MinSeong Kim and Andy 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.

[35] MinSeong Kim and Andy 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.

[36] MinSeong Kim and Andy Wellings. Applying Fixed-Priority Preemptive Scheduling with Preemption Threshold to Asynchronous Event Handling in the RTSJ. In *JTRES '09: Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems*, New York, NY, USA, 2009. ACM.

[37] MinSeong Kim and Andy Wellings. Efficient Asynchronous Event Handling in the Real-Time Specification for Java. *To Appear: ACM Transactions in Embedded Computing Systems (TECS)*, 2009.

[38] MinSeong Kim and Andy Wellings. Refactoring Asynchronous Event Handling in the Real-Time Specification for Java. In *ECRTS '09: Proceedings of the 21st Euromicro Conference on Real-Time Systems*, Washington, USA, 2009. IEEE.

[39] Harmann Kopetz. Event-Triggered Versus Time-Triggered Real-Time Systems. In *Proceedings of the International Workshop on Operating Systems of the 90s and Beyond*, pages 87–101, London, UK, 1991. Springer.

[40] Hermann Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications.* Springer, USA, 1997.

[41] Yamuna Krishnamurthy, Irfan Pyarali, Christopher Gill, Louis Mgeta, Yuanfang Zhang, and Stephen Torri. The Design and Implementation of Real-Time CORBA 2.0: Dynamic Scheduling in TAO. In *In Proceedings of RealTime and Embedded Technology and Applications Symposium*, 2004.

[42] Alcatel-Lucent Bell Labs. LTL MODEL CHECKING with SPIN. http://spinroot.com/spin/whatispin.html.

[43] J.P. Lehoczky. Fixed Priority Scheduling of Periodic Task Sets with Arbitrary Deadlines. In *Proceedings of 11th IEEE Real-Time Systems Symposium*, pages 201–209, 1990.

[44] C. L. Liu and James W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM*, 20(1):46–61, 1973.

[45] Jane Liu. *Real-Time Systems.* Prentice Hall, USA, 2000.

[46] Damien Masson and Serge 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.

[47] Sun Microsystems. Core Java J2SE 5.0. http://java.sun.com/j2se/1.5.0.

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

[49] J.K. Ousterhout. Why Threads Are A Bad Idea (for most purposes), 1996. Presentation given at the 1996 Usenix Annual Technical Conference.

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

[51] R. Rajkumar, M. Gagliardi, Sha, and Lui. The Real-Time Publisher/Subscriber Inter-Process Communication Model for Distributed Real-Time Systems: Design and Implementation. In *RTAS '95: Proceedings of the Real-Time Technology and Applications Symposium*, page 66, Washington, DC, USA, 1995. IEEE Computer Society.

[52] J. Regehr. Scheduling Tasks with Mixed Preemption Relations for Robustness to Timing Faults. In *Proceedings of 23th IEEE Real-Time Systems Symposium*, pages 315–326, 2002.

[53] 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.

[54] Herbert Schildt. *The Complete Reference Java Seventh Edition.* Mac Graw Hill Osborne, USA, 2007.

[55] Douglas C. Schmidt, Michael Kircher, Frank Buschmann, and Irfan Pyarali. Leader/Followers: A Design Pattern for Efficient Multi-Threaded Event Demultiplexing and Dispatching. In *University of Washington*, pages 0–29. Addison-Wesley, 2000.

[56] Douglas C. Schmidt, David L. Levine, and Sumedh Mungee. The Design of the TAO Real-Time Object Request Broker. *Computer Communications*, 21:294–324, 1998.

[57] Douglas C. Schmidt, Sumedh Mungee, Sergio Flores-gaitan, and Aniruddha Gokhale. Software Architectures for Reducing Priority Inversion and Non-determinism in Real-time Object Request Brokers. In *Real-time Computing in the Age of the Web and the Internet*, pages 200–221, 1999.

[58] Gert Smolka. The Oz Programming Model. In *Lecture Notes in Computer Science*, pages 324–343. Springer, 1995.

[59] S. Tucker Taft, Robert A. Duff, Randall L. Brukardt, Erhard Ploedereder, and Pascal Leroy. Ada 2005 Reference Manual. Language and Standard Libraries. *Lecture Notes in Computer Science*, 4348, 2006.

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

[61] Uppsala University and Aalborg University. UPPAAL. http://www.uppaal.com/.

[62] van Renesse Robbert. Goal-oriented programming, or composition using events, or threads considered harmful. In *Proceedings of the 8th ACM SIGOPS European workshop on Support for composing distributed applications*, pages 82–87, New York, NY, USA, 1998. ACM.

[63] Bill Venners. *Inside the Java Virtual Machine 2nd Edition*. McGraw-Hill, USA, 2000.

[64] von Behren Rob, Condit Jeremy, and Brewer Eric. Why events are a bad idea (for high-concurrency servers). In *Proceedings of the 9th conference on Hot Topics in Operating Systems*, pages 4–14, Berkeley, CA, USA, 2003. USENIX Association.

[65] Y. Wang and M. Saksena. Scheduling Fixed-Priority Tasks with Preemption Threshold. In *Proceedings of IEEE International Conference on Real-Time Technology and Applications Symposium*, pages 328–335. IEEE Computer Society, 1999.

[66] Y. Wang and M. Saksena. Scalable Multi-tasking using Preemption Threshold Scheduling. In *Digest of Short Papers For Work In Progress Session, The 6th IEEE Real-Time Technology and Applications Symposium*. IEEE Computer Society, 2000.

[67] Andy Wellings. *Concurrent and Real-Time Programming in Java*. John Wiley & Sons, UK, 2004.

[68] Andy Wellings. Multiprocessors and the Real-Time Specification for Java. In *ISORC '08: Proceedings of the 2008 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing*, pages 255–261, Washington, DC, USA, 2008. IEEE Computer Society.

[69] Andy Wellings and Alan 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*, pages 81–89, Washington, USA, 2002. IEEE Computer Society.

[70] Andy Wellings and Alan Burns. *Handbook of Real-Time and Embedded Systems Section 12: Real-Time Java.* CRC Press, USA, 2007.

[71] Andy Wellings and MinSeong 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.

[72] Matt Welsh. *An Architecture for Highly Concurrent, Well-Conditioned Internet Services.* PhD thesis, University of California, Berkeley, USA, 2002.