A Real-Time RMI Framework for the RTSJ*

Andrew Borg and Andy Wellings. {aborg, andy}@cs.york.ac.uk
Department of Computer Science, The University Of York, UK.

Abstract

The Real-Time Specification for Java (RTSJ) provides a platform for the development of real-time applications. However, the RTSJ does not take the distribution requirements of real-time applications into consideration. As distribution in Java is often implemented using Java's Remote Method Invocation (RMI), a real-time version of RMI between RTSJ implementations can provide a platform for writing distributed real-time systems.

This paper describes a Real-Time RMI (RT-RMI) framework that supports timely invocation of remote objects. The thread classes defined by the RTSJ are used to provide the client and server threading mechanisms. The memory model of the RTSJ is considered to ensure that threads correctly use memory areas and avoid memory leaks in the absence of the garbage collector. New classes are developed to control the threads used throughout the invocation and to provide new semantics for remote objects that can be invoked in a timely fashion.

1 Introduction

The Real-Time Specification for Java (RTSJ) [9] provides extensions to the Java platform to make it suitable for writing real-time applications. These extensions include amongst others the definition of real-time thread scheduling and dispatching mechanisms and a new memory model that allows the application to avoid garbage collection penalties.

The RTSJ is silent on issues of distribution. Consequently, a Distributed RTSJ (DRTSJ) Expert Group has been set up under the Java Community Process [2]. Although no draft specification has been released as yet, it is understood that the preferred programming model is the control flow model implemented using Java's RMI [11].

An initial framework by the expert group describes three levels of integration of RMI with the RTSJ [18]. At level (0), real-time Java virtual machines communicate via standard RMI. No guarantee of timely delivery of a remote request can be assumed and the programmer must explicitly pass real-time parameters with each call. This requires no extension of either RMI or the RTSJ. At level (1) integration, the notion of a real-time remote object is introduced, supported by an RMI runtime that provides timely invocation guarantees. Level (2) integration augments level (1) with distributed thread semantics.

This paper explores the facilities required by a real-time RMI (RT-RMI) and provides a framework for its implementation. The focus is on RT-RMI for level (1) integration.

We identify two requirements of RT-RMI.

- Requirement 1: Timely invocation.
- Requirement 2: Real-time support for RMI.

The first requirement aims to bound the time for an end-to-end RMI call to guarantee timeliness and bound priority inversion. Relevant distributed schedulability analysis (e.g. [17]) can then be used to guarantee the real-time properties of the whole system. The second requirement deals with extending support of RMI components such as the distributed garbage collector and serialisation with functionality for real-time environments.

This paper deals solely with fulfilling the first requirement by developing a framework for an RMI runtime that can provide timely invocation with bounded priority inversion using RTSJ facilities. We preserve the fundamental RMI semantics but extend the way remote invocations are handled in the RMI runtime. We do not provide a complete implementation but discuss how the framework could be used to develop one.

The goal of the framework is to provide an RMI foundation that is flexible enough to (1) allow an implementation to use arbitrary scheduling policies (as far as the RTSJ allows this), (2) support different real-time networks and (3) allow an application to control the real-time parameters of the invocation process. The RTSJ also requires modifications but we try to make these as conservative as possible.

An important goal is to keep separate the framework classes and the network-dependent implementation classes. This allows pluggability of the framework with RTSJ implementations and network architectures, with the framework sitting between the latter two. We achieve this by partitioning RMI as shown in Figure 1. The RMI Specification [1], though independent of any Java implementation provides no separation between threading functionality

*This work is funded by an EPSRC grant. The authors would like to acknowledge the helpful comments provided by the reviewers, in particular the paper's shepherd, Tullio Vardanega.

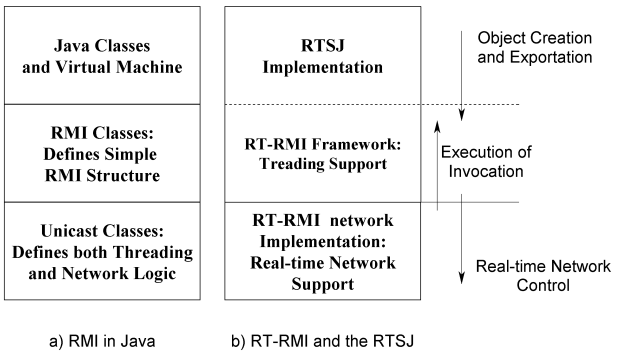

Figure 1. RT-RMI and the RTSJ

and the network (in particular the unicast classes for TCP/IP networks). Our framework provides this separation whilst supporting a stronger binding with the RTSJ implementation and exposing the thread mechanism to the application.

The rest of this paper is set out as follows. In Section 2, the classes that allow an application to interact with RT-RMI are defined. Section 3 describes the server thread mechanism for RT-RMI while Section 4 focuses on the thread properties of the client. Section 5 describes related work and finally Section 6 describes future work and concludes. A more detailed discussion of this work and an example can be found in [6].

2 Extending RMI

This section identifies the key classes in the RMI specification [1] that must be considered for the development of RT-RMI. A brief introduction to RMI and its class hierarchy is provided, followed by the RT-RMI class hierarchy.

2.1 Classes and Threads in RMI

The RMI specification defines the `RemoteObject` class that provides the `Object` semantics for remote objects. `RemoteStub` provides the client handle to a remote object while `RemoteServer` provides the server handle. The subclasses of `RemoteServer` (such as `UnicastRemoteObject`) implement the network-dependent logic.

A remote object must first be exported to allow the RMI runtime to set up the connection and prepare for remote invocations. For `UnicastRemoteObject`, the *exportObject()* methods are static to allow the export of any object that implements `Remote` and not just subclasses of `UnicastRemoteObject`. *exportObject()* returns a `RemoteStub` object that is used to bind the object to the registry. In a remote invocation, the logic in the stub packages the parameters and calls the *invoke()* method of the **ref** object. The **ref** field is inherited by `RemoteStub` from `RemoteObject`:

```
ref.invoke(Remote obj, Method method,
           Object[] params, long opnum)
```

The client threading mechanism in Java's RMI is synchronous. The RMI specification makes no restrictions on the thread behaviour on the server node. An implementation may provide a single-server invocation thread mechanism or a multi-threaded server model. It may also use some load control system such as maintaining a queue of requests and using thread pools.

2.2 Extending the RMI Class Hierarchy

The RTSJ does not attach real-time properties to regular objects but to objects that implement the `Schedulable` interface. As it is a regular object that is exported and not a `Schedulable` object, it is necessary to extend the semantics of a remote object for it to be able to export the fact that it can be run in a real-time context. The current RMI methods to invoke and export remote objects must then be extended to allow for the propagation of parameters by the client and server objects to the RT-RMI runtime.

Figure 2 shows how the class hierarchy for RMI as described in the RMI specification is extended in order to specify real-time remote objects. The new tagging interface `RealtimeRemote` shows that the object can be invoked remotely in a timely fashion. The classes `RealtimeRemoteStub` and `RealtimeRemoteServer` are analogous to their non real-time counterparts and embed the network-independent real-time functionality of the runtime. Note that we choose not to have a class `RealtimeRemoteObject` analogous to `RemoteObject` and have `RealtimeRemoteStub` and `RealtimeRemoteServer` extend this class. This approach conforms to the argument that the RTSJ does not place real-time semantics on objects.

2.3 Exporting Server Parameters

When an object is exported in RT-RMI, it may define the real-time parameters of the `Schedulable` object that is to handle the invocation. These include several parameters such as the type and scheduling parameters of the `Schedulable` object that is to handle the request, the memory area used and parameters that specify the propagation policy (that is whether to use client or server propagated policies). The propagation policy is defined and enforced by the server and can be changed at run-time.

Although the RMI documentation states that: "the functions needed to create and export remote objects are provided abstractly by `RemoteServer` and concretely by its

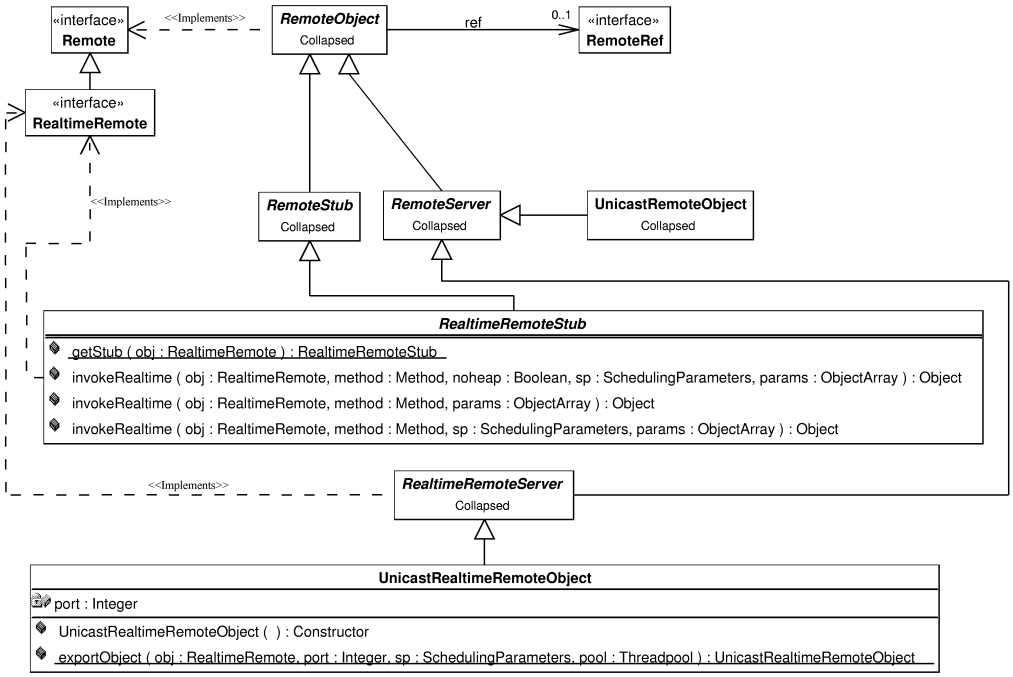

Figure 2. The Class Hierarchy for RT-RMI

subclass(es)", this class does not define any abstract *exportObject()* methods. This ensures that `RemoteServer` remains independent of any network implementation. It does not encapsulate any more semantics than those in the `RemoteObject` class it extends. The semantics of export are defined almost entirely by subclasses of `RemoteServer` such as `UnicastRemoteObject` in which the necessary logic is provided. A `RealtimeRemoteServer` object however defines new semantics which it must encapsulate: the real-time semantics of the `Schedulables` that are to service the request.

The class `RealtimeRemoteServer` should not attempt to describe the network-specific semantics used for export, be they real-time or not. Similarly, the real-time threading functionality of RT-RMI should be independent of the underlying network-specific mechanism expressed in the subclass of `RealtimeRemoteServer` such as `UnicastRealtimeRemoteObject`. A separation of concerns is required. `RealtimeRemoteServer` will provide only the real-time invocation functionality whereas subclasses (such as `UnicastRealtimeRemoteObject` that implement point-to-point TCP implementations) encapsulate network-specific information. The latter will use `RealtimeRemoteServer` to manipulate real-time properties but will not encapsulate any real-time related semantics other than those supported by the network.

An *exportObject()* operation of some class `UnicastRealtimeRemoteObject` that extends `RealtimeRemoteServer` may take real-time parameters such as `SchedulingParameters` and network-specific parameters such as a TCP port and network priority. The relevant constructors and methods are called in `RealtimeRemoteServer` to set up the real-time properties of the remote object. A thread pool can be used with each thread set to have application defined scheduling parameters before being used. The logic to listen on the port is then carried out by methods in `UnicastRealtimeRemoteObject` and related classes. A description of the relationship between these classes and `RealtimeRemoteServer`, together with a description of the `Threadpool` class is given in Section 3.

In contrast to `UnicastRemoteObject`, subclasses of `RealtimeRemoteServer` should not return a stub after export but a reference to the instance of itself or `RealtimeRemoteServer` created after export. This exposes the real-time thread mechanism to the application and allows the application to dynamically manipulate the real-time parameters attached to an exported object, for example when mode changes are required. A method to generate a `RealtimeRemoteStub` instance from a `RealtimeRemoteServer` object similar to *getStub()* in `RemoteObject` is provided in `RealtimeRemoteStub`.

2.4 Propagating Client Parameters

A client must be able to specify its real-time requirements of the server by providing real-time parameters. We introduce new *invokeRealtime()* methods in the new `RealtimeRemoteStub` class that are called by the stub instead of *ref.invoke()*. These methods obtain the real-time parameters using one of three methods described below. These parameters are packaged with the non real-time parameters into the array of parameters (**params**) that is then passed to the *invoke()* method of the `RemoteRef` instance (**ref**) of the object. The RT-RMI runtime on the remote machine retrieves these parameters, sets up the `Schedulable` to handle the invocation and calls the relevant method in the implementing object with its parameters. An implementation can distinguish between real-time and non real-time parameters at the server by either using a special separator between the two parameter types or by having all real-time parameters implement some tagging interface. None of the real-time parameters added by *invokeRealtime()* ever appear in the remote object. Note that few of the classes in the RTSJ implement the `Serializable` interface. All classes whose instance can be propagated must be made serializable. These include the `SchedulingParameters` class and its subclasses and some of the time-related classes such as `AbsoluteTime` for time-driven schedulers such as EDF.

We identify three ways by which a client can express client-propagated parameters:

- Assigning parameters to a `RealtimeRemoteStub` instance.
- Implicitly inheriting or deriving the parameters from the client `Schedulable` object.
- Explicitly specifying parameters for each invocation. For classes that extend `RealtimeRemote`, **rmic** (or its dynamic equivalent) generates extra methods for each method defined in `<ServerObject>_Stub` with appropriate real-time parameters.

The above three can be used concurrently. For example, scheduling parameters would normally use the second option whilst a flag stating whether the server is to enforce the propagation policy would use the first.

In Section 2.3 we showed how `RealtimeRemoteServer` separated the real-time functionality of RT-RMI with the network-dependent functionality defined in the subclasses. However, no subclass of `RealtimeRemoteStub` for the client is defined for RT-RMI. The reason for this is that the RMI specification uses implementation classes of the remote reference interface `RemoteRef` (such as `UnicastRef`) to define network specific properties. This is maintained in RT-RMI.

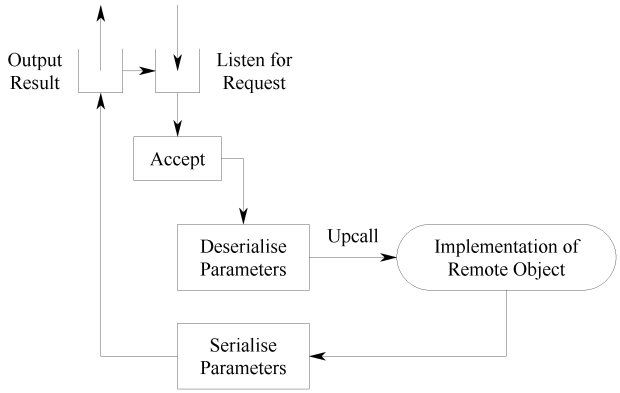

Figure 3. General Server-Side Handling of RMI

Network-specific real-time parameters are passed using one of the above mechanisms to the instance of the `RemoteRef` implementation for which the *invoke()* method is defined. The logic in *invoke()* is then responsible for extracting network related parameters from **params** and preparing the network for the invocation.

3 The Server Threads in RT-RMI

In this section, a description of the server-side functionality of RT-RMI is provided. The framework is extended with a new set of classes and the `RealtimeRemoteServer` class is revisited in further detail. We show how the `Schedulable` objects used to handle invocation requests are defined and manipulated by the application and how the correctness of the RTSJ memory model is maintained.

3.1 Control of the Threading Behaviour

Unlike RMI, RT-RMI implementations are not free to choose an arbitrary threading mechanism. We provide a specific threading mechanism that is flexible but provides a minimum functionality and semantics that an implementation must adhere to and which any application can therefore assume. This also provides a solution for a RT-RMI implementation to bound priority inversion.

Figure 3 shows how acceptance and execution are carried out in an RMI invocation with a single-threaded server mechanism. A multi-threaded server could accept the request and create/obtain a new thread to handle deserialisation and the upcall on the remote object's implementation before waiting for the next request. Alternatively it could carry out deserialisation itself before using another thread to make the upcall.

RT-RMI requires that the acceptance mechanism operate at the highest priority when client propagated parameters can be used in order to bound priority inversion. (For non priority-driven schedulers, `AcceptorRealtime` threads are the most eligible threads and are dispatched first). Once an acceptance is made, the parameters of the `Schedulable` object to handle the invocation are set by reading the values from `RealtimeRemoteServer` for server-exported parameters and by reading from the network for client-propagated parameters. This handler object will then carry out the execution of the request.

In order to allow multi-threaded servers and also to minimise priority inversion to the time taken for client-propagated parameters to be extracted, we split the server-side invocation process into two. The acceptance and extraction of the client-propagated parameters of a request is controlled by an object of type `AcceptorRealtime`. The remainder of the invocation including the retrieval and unmarshalling of non real-time parameters and the upcall to the implementing object is controlled by an object of type `HandlerRealtime`. The `AcceptorRealtime` and `HandlerRealtime` classes are detailed next.

3.1.1 The `AcceptorRealtime` Class

`AcceptorRealtime` is defined as an abstract class that implements `Runnable`, encapsulates a `Schedulable` object and is used to control and manipulate the real-time properties of this `Schedulable` object. The subclass of `AcceptorRealtime` such as `UnicastAcceptorRealtime` encapsulates the network-specific functionality of the acceptor. In particular, this subclass implements the *run()* method that contains the logic for waiting for remote invocation requests. When a new `AcceptorRealtime` object is instantiated, a new `Schedulable` object is created and **this** is passed in the constructor of the `Schedulable`. The type of the `Schedulable` can be specified at the time of export of the `RealtimeRemote` object or defaulted to either `RealtimeThread` or the type of the `Schedulable` object issuing the export. The `Schedulable` is started by running the *AcceptorRealtime.start()* method which runs *start()* on the `Schedulable` object if it is an instance of `RealtimeThread` or `NoHeapRealtimeThread`.

Figure 4 shows the `AcceptorRealtime` class and its unicast subclass. The private static methods are used to return an instance of either a heap or no-heap real-time thread depending on a boolean value passed in the constructor of `AcceptorRealtime`. (The different constructors are not shown in the figure). A similar mechanism is used if `AcceptorRealtime`'s `Schedulable` were an event handler. The static method *getNewAsyncEventHandler(t: Runnable)* returns a new event handler that is then bound to some `AsyncEvent` object passed through *exportObject()* by the application and that signifies the receipt of a request.

3.1.2 The `HandlerRealtime` and `Threadpool` Classes

Figure 5 shows the `HandlerRealtime` class and its unicast subclass together with the `Threadpool` class. Similarly to `AcceptorRealtime`, `HandlerRealtime` is also defined as an abstract class that implements `Runnable`, and encapsulates a `Schedulable` object. This `Schedulable` is used to read the non real-time section of the call from the network, unmarshall the parameters, make the upcall to the implementing object and finally marshall and send the result. The `Schedulable` object can be of any `Schedulable` type as long as the implementation makes correct use of heap and noheap versions of `Schedulable` and conforms to the RTSJ memory model.

Instances of `HandlerRealtime` may be referenced in a linked-list fashion by a new class `Threadpool` that contains a list of `HandlerRealtime` instances. When a new handler is required by `RealtimeRemoteServer`, the `HandlerRealtime` instance returned may be taken from a `Threadpool` object or created afresh, depending on what parameters were passed when the object was exported or on the defaults for that implementation. `Threadpool` instances may be shared between exported objects and serve two important functions:

- A control of the amount of concurrency allowed.
- A reusable source of Schedulable objects, thereby improving performance and predictability.

Concurrency control is achieved by having the acceptor's `Schedulable` block in `RealtimeRemoteServer` on *getHandler()* if no handler is available in the thread pool at the time.

There is a fundamental difference between `AcceptorRealtime` and `HandlerRealtime` objects that warrants there being two types of threads that handle each of the two parts of the invocation. While `AcceptorRealtime` objects are meant to live for the duration of a single exported object, `HandlerRealtime` object are reusable components shared by possibly many exported objects. Unlike `AcceptorRealtime` where the *run()* method of the subclass is the only logic required, `HandlerRealtime` defines a *run()* method itself as it must carry out some logic after the request is completed. This *run()* method calls the *handleRequest()* method of its subclass (such as `UnicastHandlerRealtime`). When *handleRequest()* returns, the *run()* methods does any necessary cleaning up and returns the thread to its `Threadpool` if it belongs to one.

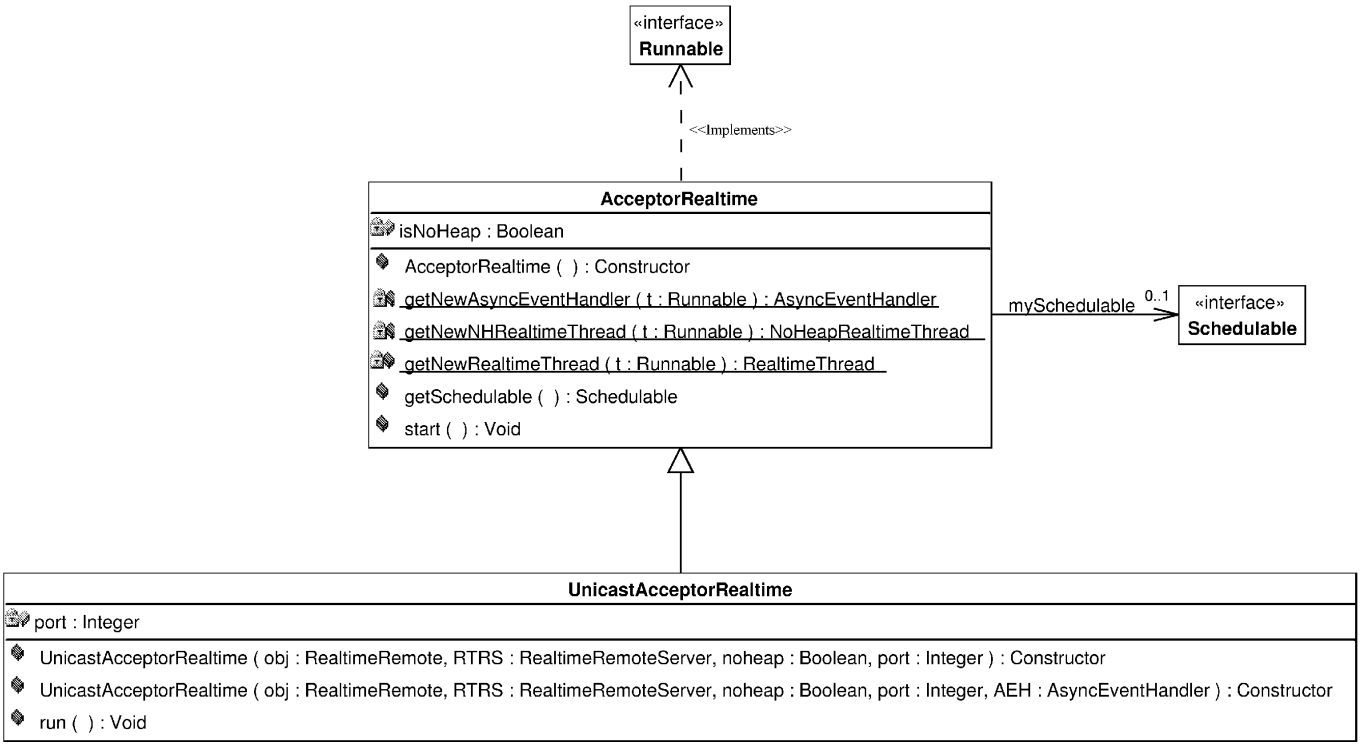

Figure 4. The `AcceptorRealtime` and `UnicastAcceptorRealtime` Classes

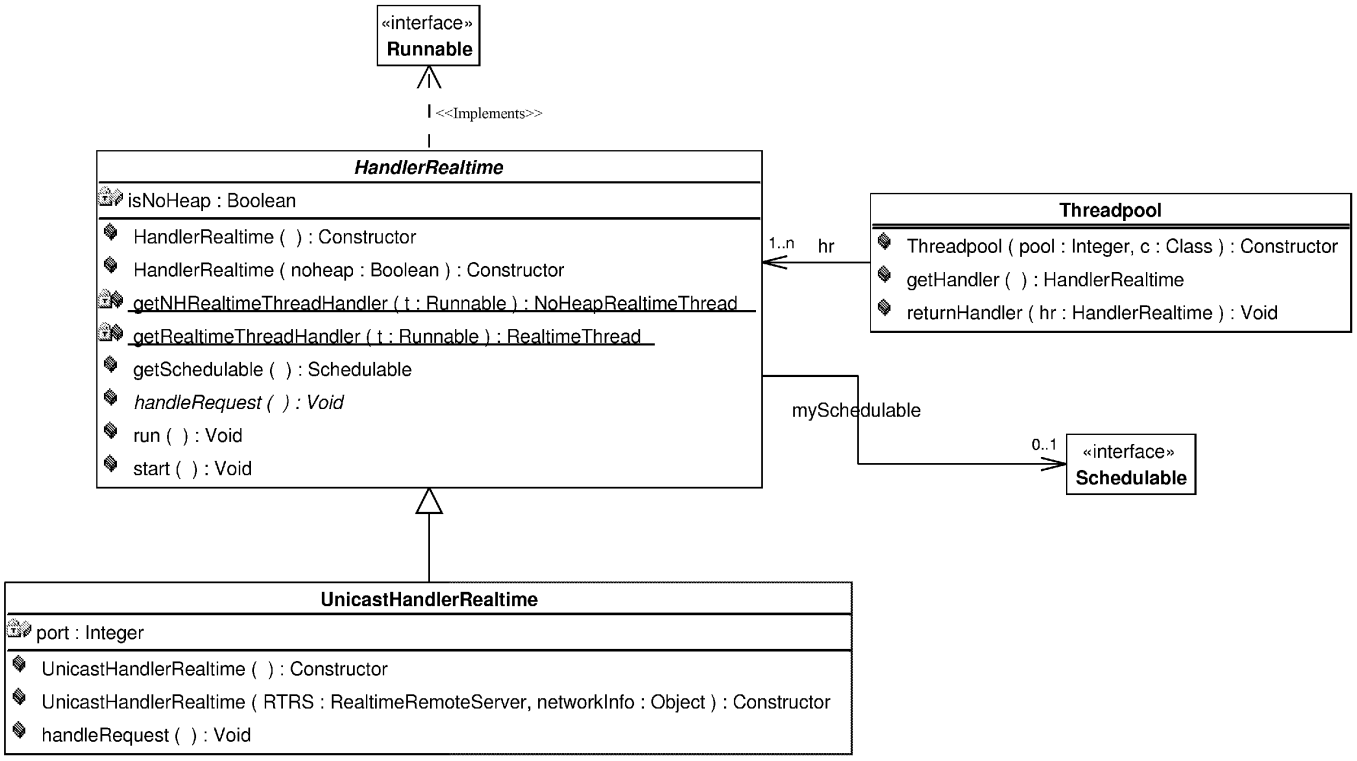

Figure 5. The `HandlerRealtime`, `UnicastHandlerRealtime` and `Threadpool` Classes

3.2 Bringing it All Together: The `RealtimeRemoteServer` Class

Section 2.3 introduced the `RealtimeRemoteServer` and described how it provides the interface to its subclasses to export objects in real-time contexts. We now show how `RealtimeRemoteServer` does this by defining a number of methods that make use of the `AcceptorRealtime` and `HandlerRealtime` classes and their subclasses as well as the `Threadpool` class.

Figure 6 shows the `RealtimeRemoteServer` class and its unicast subclass `UnicastRealtimeRemoteObject`. Each instance of `RealtimeRemoteServer` maintains a reference to an instance of `AcceptorRealtime` and to an instance of `Threadpool`. `HandlerRealtime` objects are extracted from the thread pool if one is assigned to the `RealtimeRemoteServer` object or created afresh by using the abstract *getHandler()* method. Instances of `AcceptorRealtime`'s subclass are returned by using the abstract *getAcceptor()* method.

The real-time behaviour of the request-handling mechanism is controlled by `RealtimeRemoteServer` based on the parameters passed in the *exportRealtime()* methods as called by the subclass. A final specification would define a suitable set of these methods that manipulate the behaviour of the `AcceptorRealtime` and `HandlerRealtime` objects and the real-time parameters of their `Schedulable`s.

An interesting consequence of the separation provided by the framework is that the logic for accepting and handling requests need not be fixed for all objects in the system. It is possible to pass different subclass implementations of `AcceptorRealtime` and `HandlerRealtime` dynamically at run-time. An implementation of a subclass of `RealtimeRemoteServer` may also provide export methods that take as parameters class types of `AcceptorRealtime` and `HandlerRealtime` subclasses or their instances passed by the application. Besides the added flexibility provided in allowing protocol selection and configuration, this allows an implementation to easily port and support a larger range of networks.

3.3 Memory Considerations

A number of issues arise when using the memory model of the RTSJ with RT-RMI. The primary cause of this is that different types of threads can be used to handle events which must all follow strict memory usage rules. In this section we devise a number of requirements of a RT-RMI implementation that maintain the integrity of the RTSJ memory model.

In general, the export operation may pass a number of parameters that reference objects that may exist in heap, immortal or scoped memory. Examples include references to an instance of `Threadpool` and an instance of `SchedulingParameters`. A reference to the object to export is also passed. An additional important parameter that may be passed is a boolean signifying if the `Schedulable` should be run in a heap or noheap context.

The first step of the export procedure creates a new `Schedulable` as part of the creation of the `AcceptorRealtime` object. The rules for memory areas state that this `Schedulable` inherits the memory area stack of the thread making the export. However, the new `Schedulable` might be a noheap thread and therefore must run in immortal memory or scoped memory. Moreover, it can not reference objects in the heap. Therefore a rule for exporting objects whose acceptor will run in a noheap context is that all parameters passed must not exist in heap memory or reference any objects on the heap. The application developer must therefore bear in mind that exportation is the same as creation of a `Schedulable`.

The scoping rules must be kept in mind by an application developer when an object is exported to run in scoped memory. An application that exports an object using a thread that runs in a scoped memory area may not create objects during the execution of the method in the upcall if creating that object requires any reference manipulation that breaks the scoping rules of the RTSJ. We believe that though this is the most significant burden placed on the developer it is no greater than that placed by the RTSJ on applications that make use of nested scoped memory areas.

Once the extraction of the client-propagated parameters is complete, the next step is setting up the handler. One requirement is that the `Schedulable` in `HandlerRealtime` can only be a noheap thread if the `Schedulable` in `AcceptorRealtime` is a noheap thread. This creates a small burden on the application designer to ensure this is adhered to or an exception will be raised. If the handler is to be created afresh by creating a new instance of `HandlerRealtime`, the same mechanism used above for `AcceptorRealtime`'s schedulable can be used. However, if the thread is to be taken from a thread pool and is a noheap thread, the stack of the handler must be rebuilt from the stack of the acceptor. This is a problem for a framework implementation and is transparent to the developer.

We note a final important issue that must be addressed by a RT-RMI implementation. The runtime on the server node creates a number of temporary objects such as the unmarshalled parameters sent in an invocation. If the `Schedulables` in `AcceptorRealtime` and `HandlerRealtime` use scoped or immortal memory, the garbage collector will be unable to dispose of these objects. In these cases, an implementation should nest the invocation appropriately inside scoped memory areas that are freed once a remote request is completed.

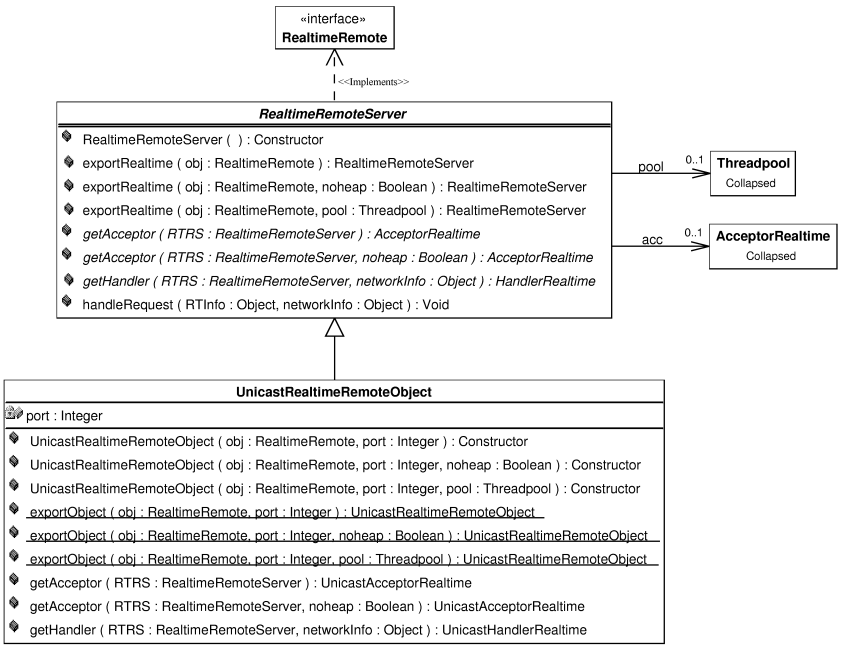

Figure 6. Detail of `RealtimeRemoteServer` and `UnicastRealtimeRemoteObject`

4 The Client Threads in RT-RMI

The client thread mechanism is much simpler to define than that of the server. A single thread is involved on the client that blocks on invocation until a reply is returned. This thread is already controlled by the application and therefore does not need to be exposed in the same way as the server threads.

Real-time applications often require more control over a remote invocation and simply blocking for an unknown amount of time in a synchronous call is not an option. There are two solutions to this which we consider: asynchronous calls and time-outs. Although there are a number of asynchronous techniques that could be used, they would modify the fundamental control-flow semantics provided by RMI. Therefore, we choose to avoid asynchronicity in RT-RMI.

Time-outs can however be used and still maintain the synchronous control-flow of RMI. They are propagated to the runtime as described in Section 2.4. There are two possible behaviors of the runtime when a timeout expires, though the way they are implemented is not defined as it depends on the support provided by the OS and the network. The first is that an exception is raised and control is returned to the application. Alternatively an event could be fired that runs an event handler propagated by the application.

4.1 Memory Considerations for Client Threads

The memory model for the client thread is much simpler as no additional threads are involved. The thread making the invocation may be a heap or noheap thread and may pass parameters that reside in heap, immortal or scoped memory. As with the server-side `Schedulables`, the invoke operation creates temporary objects that should be appropriately scoped. An object created as the return value of an invocation is created in the memory area the thread was running in at the point of invocation. No additional memory concerns are placed on the application developer.

5 Related Work

The focus in distributed real-time communication is shifting away from traditional message passing systems to more structured communication such as the control-flow paradigm as provided by remote invocation. The Real-Time CORBA Specification (RT-CORBA) [5] uses this paradigm to provide end-to-end real-time support for heterogenous real-time systems. The Dynamic Real-Time CORBA Specification [3] extends the original specification and uses a distributed thread model that allows the dynamic scheduling of the distributed thread.

There is a strong argument to utilise RT-CORBA as the distribution medium of the RTSJ. The OMG has set out an RFP soliciting extensions to the existing Java Language to IDL Mapping specification to support the RTSJ [4]. Similar considerations were made when discussing whether to use RT-CORBA for distributed real-time Ada or to extend the Ada Distributed Systems Annex [13, 10, 16]. It was high-

lighted that distributed applications would be simpler, more efficient and more reliable without CORBA. While we believe that our work can help to understand how RT-CORBA can be mapped to the RTSJ, our focus remains on RMI as the direction chosen for the DRTSJ.

The Zen project [12] seeks to develop a RT-CORBA ORB using the RTSJ. Implementing a real-time ORB in the RTSJ requires consideration of similar issues that are faced in implementing a real-time RMI runtime. An example of this is the appropriate use of scoped memory areas in the invocation mechanism to avoid memory leaks. However, though Zen uses RTSJ facilities inside the ORB, it does not specify how the RT-CORBA API would map to the RTSJ's thread and memory model.

Although there has been much work on improving RMI for general distributed systems (for example [15, 14]), there has been limited work on real-time RMI or indeed real-time distributed Java in general. We highlight the work carried out by Miguel [8] who uses a reservation-based protocol on the network to guarantee end-to-end message delivery. Miguel's work differs from ours in that we separate the real-time network from the thread handling mechanism, closely binding the latter to the RTSJ.

6 Future Work and Conclusion

Our work on RT-RMI is still ongoing and we have found several interesting problems for which the solutions are often not trivial. The memory-model of the RTSJ proves particularly challenging for RT-RMI. The second requirement described in the introduction, support for the RT-RMI, is also being considered. We are working in particular on understanding the semantics of remote references to objects in scoped memory areas to understand how the distributed garbage collector should operate. Another more ambitious area of research involves moving RT-RMI to level (2) integration. Although the distributed thread model is now well understood, even in real-time systems [7], research into new territory is required due to the new thread semantics of the RTSJ and more-so because of its memory model.

Several open problems still exist arising mainly from integrating the novel model of the RTSJ with a communication mechanism that provides a higher level of abstraction than the traditional message-passing mechanism. The RTSJ has generated a large amount of research. Achieving a distributed RTSJ is necessary as many real-time applications, particularly in embedded devices, are no longer centralised. The RT-RMI framework described in this paper provides an initial step in this direction through a flexible specification that can be adapted to any real-time network and scheduling policy of an RTSJ implementation.

References

[1] Java Remote Method Invocation Specification Available from: ftp://ftp.java.sun.com/docs/j2se1.4/rmi-spec-1.4.pdf.

[2] The JSR-50 Home Page – http://jcp.org/en/jsr/detail?id=050.

[3] Real-Time CORBA v2.0: Dynamic Scheduling, OMG Document ptc/01-08-34. *ACM Ada Letters, XXI 1*, September 2001.

[4] JCP RTSJ and Real-time CORBA Synthesis - Request for Proposal, OMG Document orbos/02-01-16. February 2002.

[5] Real-Time CORBA v1.1, OMG Document formal/02-08-02. August 2002.

[6] A. Borg. A Real-Time RMI Framework for the RTSJ, Department of Computer Science, University of York. Available from: http://www.cs.york.ac.uk/ftpdir/reports/. 2003.

[7] R. K. Clark, E. D. Jensen, and F. D. Reynolds. An Architectural Overview of the Alpha Real-Time Distributed Kernel. In *Proceedings of the USENIX Workshop on Microkernels and Other Kernel Architectures*, April 1992.

[8] M. A. de Miguel. Solutions to Make Java-RMI Time Predictable. In *Proceedings of the 4th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 379–386, 2001.

[9] G. Bollella et. al. The Real-Time Specification for Java Available from: www.rtj.org. 2000.

[10] J. J. G. Garcia and M. G. Harbour. Towards a Real-Time Distributed Systems Annex in Ada. *ACM Ada Letters, XXI 1*, pages 62–66, March 2001.

[11] E. D. Jensen. Rationale for the Direction of the Distributed Real-Time Specification for Java. In *Proceedings of the 5th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, 2002.

[12] R. Klefstad, D. C. Schmidt, and C. O'Ryan. The Design of a Real-time CORBA ORB using Real-time Java. In *Proceedings of the International Symposium on Object-Oriented Real-time Distributed Computing*, April 2002.

[13] L. M. Pinho (Rapporteur). Session Summary: Distribution and Real-Time. *ACM Ada Letters, XXI 1*, pages 14–16, March 2001.

[14] J. Maassen, R. van Nieuwpoort, R. Veldema, H. E. Bal, and A. Plaat. An Efficient Implementation of Java's Remote Method Invocation. In *Principles Practice of Parallel Programming*, pages 173–182, 1999.

[15] C. Nester, M. Philippsen, and B. Haumacher. A More Efficient RMI for Java. In *Java Grande*, pages 152–159, 1999.

[16] S. A. Moody (Rapporteur). Session Summary: Distributed Ada and Real-Time. *ACM Ada Letters, XIX 2*, pages 15–18, June 1999.

[17] K. Tindell and J. Clark. Holistic Schedulability Analysis for Distributed Hard Real-Time Systems. *Microprocessing and Microprogramming - Euromicro Journal (Special Issue on Parallel Embedded Real-Time Systems)*, 40:117–134, 1994.

[18] A. Wellings, R. Clark, D. Jensen, and D. Wells. A Framework for Integrating the Real-Time Specification for Java and Java's Remote Method Invocation. In *Proceedings of the 5th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 13–22, 2002.