# An RTSJ-based Reconfigurable Server Component

Mohammed Alrahmawy
Department of Computer Science
University of York, York, UK
mrahmawy@cs.york.ac.uk

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

## ABSTRACT

The Real-time Specification for Java provides predictable memory and scheduling models for developing real-time systems using the Java language. However, it is silent on providing communication mechanisms suitable for distributed real-time systems. In this paper we define a synchronous and asynchronous communication component model to support different synchronous and asynchronous services and show how this model can be integrated with Java RMI in order to provide high predictability and better performance.

**Categories and Subject Description**
D.1.3 [**Concurrent Programming**] Distributed Programming; D.2.11 [**Software Architectures**] Patterns (client/server)

**Keywords**
Realtime, Java, RTSJ, Component, Configuration

## 1. INTRODUCTION

Distributed systems are now an indispensible technology for many industrial and commercial applications. Hence, many programming languages provide integrated communication and networking mechanisms to ease the development of distributed software systems (for example Java and Ada). However, although many industrial and commercial sectors (e.g. defense, nuclear, chemical, …etc) have a high demand of real-time and predictable performance, most of current communication and networking technologies have been designed and built without, or with a limited, consideration of supporting real-time behavior.

The Java programming language is one of those languages that have provided very efficient communication and networking mechanisms (e.g. RMI, sockets, etc). This support, along with Java's strong semantics and object-oriented programming model, has resulted in Java being one of the first choices for distributed software designers and developers when building highly efficient non-real-time distributed systems.

However, in general, Java has not found the same success in building real-time systems; this is due to the lack of support of predictable memory and scheduling models. Therefore, the Real-Time Specification for Java (RTSJ), (originating from JSR001 [1]), has been proposed to provide the required extensions necessary to be integrated with the Java platform to provide more predictability.

Unfortunately, the RTSJ was developed mainly for non-distributed systems. Hence, there was no consideration of providing a real-time communication models within it. JSR050 [2] was launched to address this limitation and provide a distributed real-time Java specification (DRTSJ). Unfortunately, work in this specification seems to have stalled and there is no publication from it since 2007.

In this paper we present the design of a component that provides both synchronous and asynchronous communication that is integrated with the RTSJ memory and scheduling models. We also show how Java's RMI approach can be integrated with our proposed component model to provide a proactive asynchronous RMI model. In the following sections we first present the research work related to ours, then we discuss our own proposed model, and finally we provide our conclusions.

## 2. RELATED WORK

Java has been widely used for developing distributed and non-distributed systems. However, due to its inherent unpredictability, it has not been used to develop real-time systems, neither distributed nor centralized. The RTSJ provides a predictable memory model that uses scoped memory areas to avoid the unpredictability due to the use of garbage collector. Also, RTSJ defines a scheduling model that provides an integrated real-time scheduler and predictable schedulable objects. However, the RTSJ targets only centralized systems with no consideration of distributed systems. Therefore, in order to add real-time distribution support to the RTSJ, researchers in [3, 4] have chosen the Java RMI facility as the communication mechanism that has to be extended to support real-time distribution. They presented a framework for integrating the RTSJ and Java RMI to be a basis for building Distributed Real-time Specification for Java (DRTSJ). In their proposed framework, they provided three levels of integration between RTSJ and Java RMI. These integration levels interpret the different possible implementation and programming models of the client and the server participants of an RMI call. In the following section we discuss related work in the context of these integration levels.

### 2.1 Level (0) integration

This is the minimum level of integration where Java/RTSJ objects communicate remotely using non-real-time RMI. This requires no modifications or extensions to either RMI or to the RTSJ. In this level, the proxy at the server side is implemented as a normal Java thread, while the client can be either implemented as a normal Java thread or even as a real-time thread. Therefore, in this level, clients call the remote methods without expecting a timely delivery of their calls. So, it is the developer responsibility to pass any scheduling or release parameters.

## 2.2 Level (1) integration

At this level of integration, the Java/RTSJ objects communicate remotely using real-time RMI. Hence, this level of integration assumes a change in the RMI programming model and implementation by defining a `RealtimeRemote` interface, which extends the `java.rmi.remote` interface, in order to be used by objects to be exported as real-time remote objects. When objects extending this interface are exported, they have a real-time RMI structure that is implemented using RTSJ, where the proxy at the server side of the call is implemented as real time thread (or no-heap real-time thread for stronger real-time guarantee to avoid the use of the heap memory for better predictability). Hence, the scheduling and release parameters of the RTSJ schedulable objects at the client side, or default parameters for normal Java client objects, can be propagated through the RMI protocol and assigned to the server's proxy to ensure timely predictable delivery of RMI requests. The programming model used at this level is similar to the one provided in [5] to provide predictable RMI using reservation-based scheduling techniques.

Another example of research that has been proposed using this level of integration was presented in [6], where the authors proposed an RTSJ-based framework for extending RMI. In their model they used thread pools as a source for real-time call-handlers; a mechanism that limit the concurrency of server-side threads Moreover, in order to guarantee timely predictable invocation of remote method calls, the authors assumed a set of modifications to the RMI internal structure to enable propagating the client's scheduling and release parameters to the server.

A further example of using this level of integration was presented in [7, 8], where the RMI internal structure has been changed to use RTSJ schedulable objects as listening and worker threads that are responsible of handling the remote calls at the server side. Here, a server centric approach for preserving real-time constraints has been adopted instead of propagating the real-time constraints from the client to the server, in order to provide full component isolation for the exported objects.

A final example of using this level was presented in [9], where two different models of RMI have been proposed:
**(a) Safety critical RMI.** This model, which is a direct implementation of level 1, is based on the Ravenscar-Java profile [10]. Where two execution phases are assumed: (i) *initialization phase*, e.g. creating a pool of object that holds a set of reusable schedulable objects as acceptors and handlers to be ready for handling clients' requests during the mission phase, and (2) *mission phase* where the application is executing the client invocations themselves.**(b) Soft Real-time RMI.** This model adopts the reservation-based model provided in [5] where the server–side threads of the RMI are extended to support the concept of sessions. In this model, once a listener thread accepts the client request to make a session, an acceptor thread is bounded to this session. Hence, any following request from the same client is directed to this acceptor thread that hands it out to a handler.

Also, to support their models, the authors presented in [11] a proposed model for a predictable serialization to overcome the unpredictability inherited in normal Java object serialization due to the language support for dynamic class loading.

## 2.3 Level (2) integration

This is a more general form of integration where the Java/RTSJ objects communicate remotely using a distributed real-time thread model. This level aims to extend Level 1 mentioned above to support distributed real-time threads in order to avoid the deadlock possibility inherent in levels 0 and 1. The distributed thread model [12] proposes that the thread has a unique system-wide identifier and its locus of control can move freely across the distributed system by calling methods on remote objects, where the thread is eligible for execution at only a single site of the distributed system, i.e. on the site that is hosting the remote object that is encapsulating the current remote method call made by this thread. To support this model, the authors assume that RMI architecture is to be extended to enable clients to extend the `DistributedRealtimeRemote` interface in order to be exported as real-time remote server objects. Furthermore, the RTSJ threading model is assumed to be extended to include a definition a `DistributedRealtimeThread` to be used for implementing the distributed real-time thread, where `DistributedRealtimeThread` extends `Remote-Thread` interface that defines two sets of methods. The first one is capable of both serializing scheduling and release parameters among the sites to where the distributed real-time thread is moving to/from, and the other set of methods are managing the state of the distributed thread, i.e. starting and interrupting its operation.

The distributed thread model was adopted as the core element required of the DRTSJ in [2,4]. The work in this specification is very slow, and may have been suspended as the last publication available to us was an early draft of the specification published in 2007.

In the following sections we present a review of the three server models, used in the above approaches: multithreaded synchronous, reactive asynchronous, and proactive asynchronous. In this review we will present the basic architecture of each model followed by a discussion of its pros and cons. Next, we present our proposed model of a server component based on the RTSJ that can be configured to run as one of the server models. Then, we show how we can integrate this component within the RMI architecture in order to provide a reconfigurable RMI server object.

## 3. SERVER-SIDE I/O DESIGN PATTERNS

To be able to communicate with the client, the server should provide a set of basic operations including accepting client connections, receiving the request, decoding it, processing it, and finally returning the result to the client. Different software patterns can be used to provide the integration and handling of these operations in one consistent model. In this section we review three of the software patterns. These three server-side patterns provide different server-side I/O networking communication mechanisms and scheduling models for handling user requests.

### 3.1 Multithreaded synchronous server

In this type, the calling thread blocks waiting for the result of the execution to return back from the server. This is the most common pattern used for designing many software systems, e.g. the Java RMI remote object implementation. The *acceptor-handlers* pattern [13], which is an example of this category, has mainly the following elements:

**(1)** *Acceptor Thread:* This thread blocks monitoring a network endpoint waiting for connection requests coming from clients. Once a request is received, this thread initiates another thread, a handler, to synchronously react and handle the request while the acceptor resumes monitoring the end point waiting for other connection requests.
**(2)** *Handler Thread(s)*. This is created, or initiated, by the *acceptor* thread to synchronously handle the request(s) received and return its results, if any, to the client.

Although this pattern is very simple, it is not scalable and not efficient for high performance I/O required by many real-time systems due to the unbounded nature of the pattern. The pattern in this form will need the server to be able to create as many dedicated handler threads at the server side as the number of concurrent calls arriving to it, which makes it inefficient for handling high number of concurrent requests as [14, 15]: (1) Some operating systems do not provide threading facilities. (2) The high concurrency-overhead (e.g. context switching, synchronization, and cache coherency management). (3) The requirement for coordination among threads accessing server shared resources in order to prevent race conditions, and (4) dependence on the physical limitations of the server, e.g. memory, networking capacity, and processing resources. This in turn can affect the predictability of client calls, as a high number of concurrent calls on the same server over its physical capacity will enforce the delay of even the rejection of the clients' requests. Furthermore, as the client requests are blocked and not reusable during call execution at the server, they are considered as wasted resources until receiving the call result. This can be very sensitive problem for many real time systems with very limited resources.

Some variations of this design patterns can provide enhanced performance to make the synchronous blocking I/O pattern applicable in real-time systems. These patterns reuse handler threads from a thread pools in order to limit the degree of concurrency allowed at the server in order to have a predictable execution time of the requests. Moreover, the thread pool mechanism allows multiple threads to coordinate themselves and protect the critical sections during the receiving and executing the requested calls. A common example of such enhanced patterns is the *leader follower* [14] pattern, in which only one thread, *the leader,* at a time is blocked waiting for receiving client requests. Meanwhile, other threads, *followers*, are queued up waiting for their turn to be the next leader. Once the *leader* thread receives a request from the client, it firstly notifies the thread pool to promote one of the *followers* threads to be the next leader. Then, it starts to act as a handler thread to handle the client request. Once, the handler finishes processing the requested call, it reverts back as a *follower* thread in the thread pool. In this manner, multiple, but bounded, number of handlers can handle clients' requests while only a one leader is waiting for the next request.

## 3.2 Non-blocking synchronous server

In this category, the calling thread does not block-waiting for the call to be finished. Rather, the invoked system immediately returns either the result of the execution, if it was able to process it. Otherwise, it returns an acknowledgement to the caller that the call cannot be processed. Hence, it is the responsibility of the caller to remake the request later, if required, or just ignore it.

An example of a server belonging to this category is the reactive server whose design is based on the *reactor* design pattern presented in [15]. This pattern has the following elements:
(1) *Handles.* To identify resources managed by the OS, e.g. socket endpoint of a network connection.
(2) *Synchronous event dispatcher*. Blocks monitoring events occurring on the handles. , e.g. accept connection, read, or send data request. Once an event occurs on one of the handles, it notifies the initiation dispatcher to react to it.
(3) *Initiation dispatcher (Reactor).* Define an interface for registering, removing, and dispatching event handlers associated with the events that occur on handles. Once it is notified by the synchronous event dispatcher of an event occurrence, it triggers the event handler associated with this event.
(4) *Concrete Event Handler.* Defines a set of methods that represent the operation to be executed when a certain event occurs on one of the handles, Event handlers are responsible for writing the return result, if any, to the client and sends this result in a non-blocking mode, i.e. if the client is busy and cannot receive the result, the write operations returns immediately with an acknowledge of blocking possibility, hence the operation can be repeated later to avoid blocking the server thread.

As the principal of work of this pattern is the direct reaction to events registered within the systems, all these software elements can be running within the context of a single *reactor* thread. Therefore, this server model acts as a single threaded server and it offers the following benefits [15, 16]:

- **Portability of the design among many operating systems**. As it does not need multi-threading, it can be built on any operating system.
- **Low concurrency overhead**. As there will be no context switching, nor synchronization, as it is using single threaded model.
- **Modularity**. As it decouples the application logic from the dispatching mechanisms.

However, the reactive server pattern has the following set of drawbacks [16]:

- **Program Complexity.** The server logic can be very complicated to avoid blocking the server during handling client requests.
- **Less efficiency for multithreaded systems.** As it adopts a single threaded model, it cannot utilize the hardware parallelism effectively. Hence, it is less efficient for servers on multicore hardware.
- **Has no use of the predefined system schedulability.** Operating systems of multithreaded architectures supporting pre-emptive threads are responsible for scheduling and time-slicing the runnable threads onto the available CPUs. This scheduling support is not useful for a single threaded server mode. Hence, it is the developer responsibility to carefully time-share the thread among all clients communicating with the server. This can be possible only for requests that require non-blocking operations with short duration.

## 3.3 Non-blocking asynchronous server

In the non-blocking asynchronous pattern, the control immediately returns to the calling thread reporting that the call has been delivered to the called system. The called system resources, e.g. using kernel threads and system buffers, handle the call. Then, when the result of the call is ready, the called system notifies the calling thread, e.g. using a call back method. Hence, the calling thread can retrieve the result of the call. As the call is handled by a called system on behalf of the calling thread, the calling thread can be reused to do some other processing during the call execution. This pattern requires some supporting facilities from the operating system capable of performing true asynchronous operations on behalf of the calling thread.

The *Proactor* design pattern [16] is a non-blocking asynchronous pattern. The key elements of this pattern are:

(5) *Proactive Initiator*. This is the entity of the server application that initiates the asynchronous operation and registers with it both a completion dispatcher, and a completion handler to be notified when the asynchronous operation completes.

(6) *Completion Handler(s)*. To be notified by the completion dispatcher to start execution when the associated asynchronous operation is completed.

(7) *Asynchronous operations*. These are the operations to be executed by the operating system on behalf of the server application.

(8) *Asynchronous operation processor*. This is the operating system implementation responsible for executing the asynchronous operation, and notifying the completion dispatcher when finished.

(9) *Completion Dispatcher*. This is responsible of monitoring the completion events of the asynchronous operations executing by the asynchronous operation processor. Once notified of a completion of an event by the asynchronous operation processor, it calls back on the completion handler associated with the completed operation to start execution.

Servers built using proactive patterns offer a set of benefits over the multithreaded or reactive servers, these benefits include:

- **Higher level of separation of concerns.** In this pattern, two decoupled groups of operations are defined: application independent asynchronous operations, and application-specific functionality operations. Hence, each group can be built as reusable configurable component to perform the required level of service.
- **Better application logic portability.** As mentioned above, the decoupling of asynchrony operation from event dispatching operations, help to build reconfigurable components which make it more portable to work on different platforms and operating systems,
- **Encapsulation of concurrency within the completion dispatcher.** In this pattern, the completion dispatcher can be configured with several concurrency strategies independent of the number of concurrent requests, e.g. it can be configured to run as a single threaded, unlimited multithreaded, or limited multithreaded using thread pools.
- **Decoupling of threading policy from the concurrency policy.** As the asynchronous operation processor executes the asynchronous operations on behalf of the proactive initiator, the server will not need to spawn new threads to increase concurrency if a lengthy asynchronous operation is to be executed. Hence, the server can assign a concurrency policy different from the threading policy. For example, in a multiprocessor server, the server can be configured to use a single thread for each CPU, but it can service a higher number of clients simultaneously.
- **Higher performance.** The call-back notification mechanism, used by the asynchronous operation processor to notify the completion of asynchronous operations, enhance the system performance as no logical application thread will be blocked waiting for operation completion. This in turn will minimize the number of concurrent threads running to be equal to the number of completed operations, which in turn minimize the context switching time and the required system resources.
- **Simpler application synchronization model.** As the completion handlers use the asynchronous operations instead of spawning additional threads, the synchronization and concurrency required among the server application elements is minimized.

With all these benefits, the proactor pattern has two major drawbacks:

- **It is hard to debug.** Due to using a call-back mechanism from the operating system, it is difficult to trace the flow of the execution to find out sources of errors.
- **Proactive control *may* need to have a control over the order of execution of outstanding asynchronous operation**, so the asynchronous operation processor must support efficient scheduling facilities such as, prioritizing, termination, etc.

## 4. A CONFIGURABLE SERVER-SIDE COMPONENT

As the current RTSJ is silent about the networking communication, in the following, we present a proposal for designing a general slave-server communication component model based on the predictable memory and scheduling model provided by RTSJ. The server component is assumed to provide a set of reconfigurable properties from which the programmer can configure the scheduling and concurrency policy of his own server.

Firstly, we present the basic elements that form our component, then we will discuss how it can be build using the RTSJ and how the external interface of the component can be used to reconfigure it to adapt to the environment in which it is to be used.

### 4.1 Internal structure

In the design of our server-side component we assumed that the component should provide flexibility in configuration so it can support as many as possible of the server scheduling and concurrency models discussed above. Hence, we adopt the *Proactor* design pattern as a basis of our design as it is the most flexible (and can even be used to emulate the other models). In our design we assume the component has the following elements.

(1) **Proactor dispatcher**. Both of the proactor initiator and the completion dispatcher are combined and integrated within this element. This element is responsible of initiating the processing operation either synchronously or asynchronously according to the synchrony policy configuration, i.e. it creates synchronous operation for reactive servers, and asynchronous operations for proactive servers. Also, it registers with the operation a completion dispatcher and a completion handler. Then, when an event notification arrives from the selector, this element acts as an

event firer, in order to start an event completion handler either for the accepted connection or the incoming request. Where the component' policy of handling requests can be configured to allow the acceptor to define different forms for the creation of the handlers, e.g. a handler/client or reusable handlers from a thread pool.

(2) **Channels**: these are the communication endpoints used by the component; these channels can be classified as server channels or client channel. These channels are not configurable by the user. However, they are affected by the configuration of the component's synchronization policy, i.e. the channel can work in blocking mode, when the server is configured to be synchronous, or non-blocking, when the server is running as reactive or proactive.

(3) **JVM/OS Asynchronous Operation Processor**. This is the interface provided by the JVM to forward processing of the asynchronous operation to the operating system.

(4) **Selector.** This element blocks waiting the notifications coming from the JVM/OS, to indicate the occurrence of certain predefined events occurring at the channels, in order to notify the proactor dispatcher to react to these events.

(5) **Request Handler Logic Runnable**. These are the set of completion handlers operations executed by the server to react to the client requests. These operations can be executed either synchronous or asynchronous according to the component synchrony policy. Moreover, since there are four events defined on the channels monitored by the server, i.e. connect, accept, read, and write, then for the server component we can define up to four different *Request Handler Logic Runnable*, i.e. ConnectorHandler, AcceptorHandler, ReaderHandler, and WriterHandler.

(6) **Executors.** These are elements that are responsible of executing the logic defined by the user, i.e. *Request Handler Logic Runnable*, to process the required server processing.

**(7) Executors' Pool.** This is an optional element that manages a fixed preconfigurable size of a reusable set of executors. It is responsible of controlling the concurrency model of the component.

## 4.2 Java I/O for network communication

Java provides support for network I/O operations through two main packages:

- `java.io`: This package holds the *original* classes and interfaces to manipulate networking communication and serialization through data streams. Hence, this package supports only the use of the multithreaded synchronous server model. Where clients are using mainly blocking synchronous calls, through using instances of the `java.net.socket` whereas the server waits for the client requests through an instance of `java.net.ServerSocket` class.

- `java.nio`: This package, first introduced in J2SE 1.4, holds the new I/O classes that was a result of the JSR051 [17] in order to support non-blocking communication mechanisms. The main addition of this package, for networking, is extending the Java sockets to support the new abstraction of *I/O Channels* that are capable of transferring data between sockets and NIO buffers, where NIO buffers are buffers that occupy the same physical memory used by the operating system for native I/O calls. This has been done through the introduction of the classes `java.nio.channels.SocketChannel` that support the non blocking selectable reading and writing operations over the `java.net.socket` class, and the `java.nio.channels.ServerSocketChannel` which supports accepting asynchronously selectable calls from clients. The second important addition for networking in Java NIO was the provision of selectors classes, subclasses of the abstract class `java.nio.channels.Selector` class, which provide a very efficient mechanism for multiplexed non-blocking I/O facility over channels by using the lower level operating system facilities for writing scalable server. Hence, for our component model, this package is essential for writing the proactive server model.

It should be mentioned here that a set of enhancements to complete the asynchronous model, e.g. returning future object for pending results, is supposed to be provided by the JSR201 and to be added to the NIO, called NIO2 [18].

## 4.3 Component configuration

In this section, we present the configurable properties of the server component.

(1) **Proactor dispatcher.** This acts as the main element of the component. It can be simply implemented as a passive component using RTSJ with the following configurable properties:

- **Network Address, Port No**. These two properties together define the endpoint on which the server component will receive the requests for communication.

- **Synchrony Policy**. This defines the synchrony policy of the server. It should have one of three values (1) *Procative*, (2) *Reactive*, (3) *Synchronous*.

- **Memory Context.** This defines the RTSJ memory area in which this component object is to be created. The choice of the correct memory area type to be assigned to this property is dependent on the nature of the server and its lifetime. For servers with no real-time requirement, the heap memory would be the best choice. However, for real-time component with predictable memory management, the choice can be either immortal memory for servers with life time duration equal to that of the system otherwise, a scoped memory area (LTMemoryArea, or VTMemoryArea) would be the best choice for servers with shorter lifetime duration.

(2) **Selector.** This element is directly dependent on the JVM of the underlying operating system asynchrony support. Hence, to enhance the portability of the component, the following property is important to be configurable:

- **Selector Type**. As different operating systems have different asynchrony mechanisms, and even within the same operating system there could be more than one of such mechanisms, e.g. in Linux, there is asynchrony support using poll, epoll, ...etc. Hence, it is necessary for the JVM to have different implementations of the selector using these mechanisms. Hence, the developer can select one of these implementations, by selecting a Java class that extends the `java.nio.channels.Selector` to offer a Java interface of this implementation.

(3) **Executors.** These are the elements responsible of executing the call handling logic, implementing them in RTSJ can be reconfigurable by the following:

- **Concurrency Control**. This value of this property can be assigned only in case of using either *Proactive* or *synchronous* synchrony policy. It defines the way in which the executors are to be implemented and managed within the component. The value of this property can be set either as *Fixed* or *unlimited.* Where the executors in the former case are reusable schedulable object retrieved form a pool of reusable schedulable objects, while in the later case they are implemented as a dynamically created schedulable object, i.e. handler per client.

- **Concurrency Limit.** This value is active only when the value of the *Concurrency Control* is set to be *Fixed*. This value defines the size of the reusable object pool of the executors' schedulable objects.
- **Scheduling and Release Parameters Assignment.** This is an important property for controlling the predictability of executors handling user requests that are configured to be running as schedulable objects. These parameters can be configured either:
- On a per method basis, i.e. different methods within the same calling object have different levels of priority,
- Or, on a per component/object basis, i.e. the priority of the call is dependent on the calling component/object,
- Or, on a client machine or program basis, i.e. the priority of the call is dependent on the importance of the calling client machine/program.

There are different approaches for assigning the scheduling parameters of these executors as follows:

a. **Server Centric approach**. In this approach, the server is responsible of holding the necessary **information** to assign the parameters, e.g. by loading them from predefined static parameters tables.

b. **Client Propagated approach**. Where, the server receives these parameters with the call request from the client.

Although, the second approach is more dynamic, it may require the user of the component to know exactly the protocol used internally by the component to receive the client propagated information.

**(4) Request Handler Logic Runnable.** This holds the logic of services provided by the server. The logic is executed by executors, which in our model can be executed by schedulable objects. However, as proposed in [19], an executable logic running by schedulable objects can be represented in RTSJ as an encapsulated method. Hence, to be externally assignable by the developer in order to be executed by the executors the developer writes his own encapsulated method and assigns it to this property of the component.

## 4.4 Component implementation

The sequence diagrams in Figure 2, show a prototype implementation of the proposed component using jRate on Linux platform. As shown in the diagrams, the operation of the component runs in two phases: *initialization* phase and *execution* phase.

### *4.4.1 Server initialization phase*

In this phase all the objects that will be available for the lifetime of the server component are created:

(1) The *proactor dispatcher* constructor uses its configured network address and port to create internally an instance of the ServerSocketChannel class that will monitor the incoming requests to the server. Then it sets it to run in either blocking mode (in case of multi-threaded server) or non-blocking mode (in case of proactive or reactive server). Furthermore, if configured appropriately, it creates the executors' pool, with the configured number of RTSJ executors, i.e. schedulable objects. Where these executors are configured to be real-time threads, asynchronous event handlers (AEH), no-heap real-time thread (NHRT), or even NoHeapAEH, where for AEH and NoHeap AEH an event object is created for each handler to trigger its execution. All these objects are created within the memory area configured as a memory context for this component.

(2) The *Selector* is created using one of sub classes of the Java NIO `java.nio.channels.Selector` class. For example, in Linux we can use the sub class EpollSelector class as an assigned value of the *SelectorType* property. Hence, this class can be used to create the selector demultiplexer by calling the method `Selector.open()` on it.

(3) The *proactor dispatcher* registers the accepted request event at the server channel to be handled by the selector.

### *4.4.2 Server Execution Phase*

Once the server has started its execution phase, by calling the start *method ()* of the *proactive dispatcher*, it enters an infinite loop to handle incoming client requests. Once a client request a connection to this server, the operating system notifies the selector to react to this event. The selector in turn, sends a server key, an object holding event data, to the *proactor dispatcher* to process the request. If the proactor dispatcher can accept the request, a socket channel is created to communicate with client invocations.

After this stage, the connection acceptance, the server component behaves differently for each type of the supported server types as the following stage involves the operations of receiving, decoding, and handling the client requests. Next we discuss the different processings for each of the three considered server models.

**(1) Asynch Proactive Model Execution Phase.**

This model, shown in details in figure 1, runs as a non-blocking asynchronous model; once the connection is created, the operating system notifies the selector of the occurrence of this event, In turn the selector notifies the proactor dispatcher and forwards to it the event information and attaches with it a socket channel object (client channel) responsible of future communication with this client. The proactor dispatcher configures the client's assigned socket channel to run in non-blocking mode, and then it registers both its read and write events with the selector in order to be notified when any read or write operation is performed on this channel. Then, the proactor dispatcher retrieves a free executor from the executors' pool and assigns to it the configured *acceptor handler* in order to be a completion handler of the acceptance event. As the acceptor handler executor will run before receiving any data from the client then, the scheduling and release parameters of this executor cannot be propagated from the client to the server. Therefore, scheduling and release parameters of the acceptor executor can either have default values, or it can be loaded from static tables, i.e. only the server centric approach is supported for acceptor executors.

One possible use of the acceptor handler is to retrieve the client-propagated parameters for user requests in order to support the client propagated parameters approach for the next client requests. When a request arrives from the same client, a read event is fired on its corresponding client's socket channel at the server side; this event is delivered through the selector to the proactor dispatcher. Then, the proactor dispatcher retrieves another free executor from the executors' pool. However, this time the executor can be assigned either server centric parameters or client propagated parameters, if it is retrieved by the acceptor handler. Then, the executor starts executing the configured *Reader Handler* of the component.

Once the server starts to write bytes to the registered client socket channel, an event is fired by the operating system and propagates to the proactor dispatcher. Then, the proactor dispatcher retrieves another executor from the executors' pool and assign its parameters in the same manner as in the read event. However, in this case the executor will execute the logic defined in the Writer handler runnable.

Then, the *proactor diapatcher* retrieves a reusable free executor form the executors' pool. This reusable executor is initiated in a scoped memory area so that this is the memory allocation context of objects created during the executor's execution, and this memory is reclaimed back after finishing the executor's execution. The executor is responsible of executing the *Request Handler Logic Runnable* corresponding to the completion handlers of events occurring at the channels. Since the system can be configured by the developer to process the client request. Therefore, before starting execution, the executor parameters has to be initialized; these parameters include the proactor dispatcher object, and the socket channel created for the client to be serviced. Also, at this stage, before starting execution, scheduling and release parameters are to be retrieved and assigned for the executor. These parameters are retrieved either from the parameters tables, in the case where the component is configured to use server centric approach, or it is retrieved from the client and assigned to the executor dynamically. The executor is reserved for the client but it does not start execution until a request is received.

**(2) Non-Blocking Synch Reactive Model Execution Phase.**

The execution phase of this model, see figure 2, behaves initially the same as the proactive model, i.e. the dispatcher is notified by the operating system through the selector of the incoming events on the channel, which is running in a non blocking mode. The major difference between the reactive and proactive model is in the way of executing their completion handlers. As the reactive model is assumed to be single threaded, when an event occurs on the client channel, instead of reusing an executor from the executors' pool to run the acceptor handler, reader handler, or writer handler, the proactor dispatcher itself acts as the executor of all the handlers logic. According to this, the proactor dispatcher enters the scoped memory assigned for this handler and runs its logic. Scheduling and release parameters can be retrieved in the same way as defined in the proactor pattern. However as only one executor is running in this model, the scheduling and release parameters of this executor will have to be changed frequently and dynamically each time a new event arrives to the system

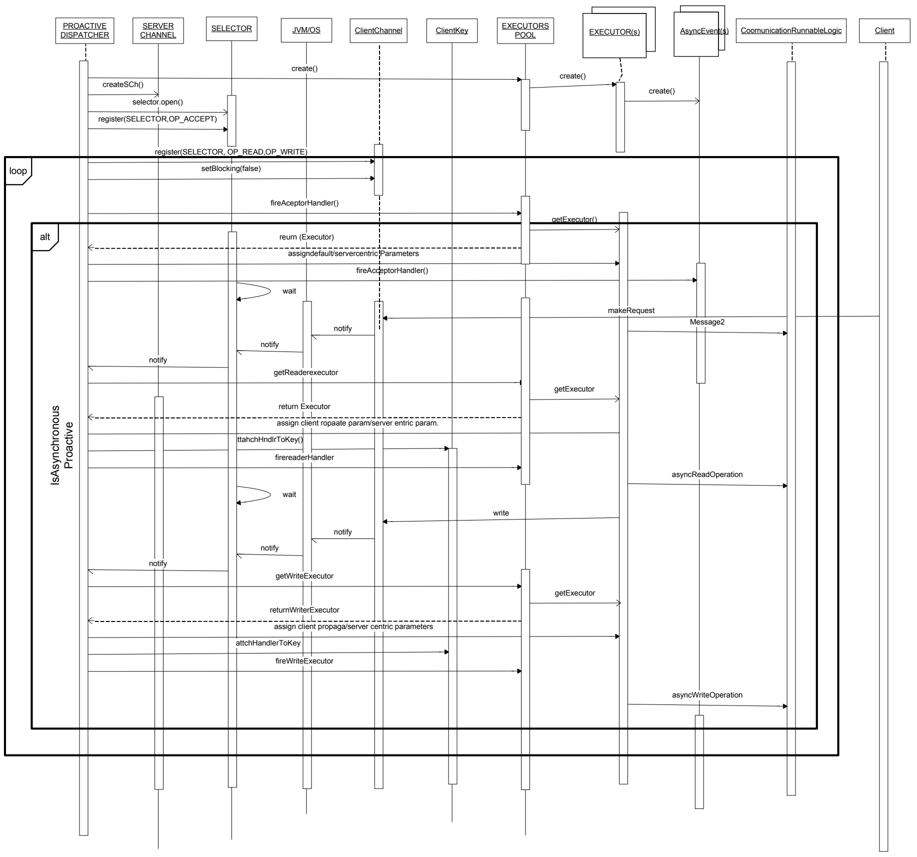

**Figure 1: The Asynchronous Proactive Server Model**

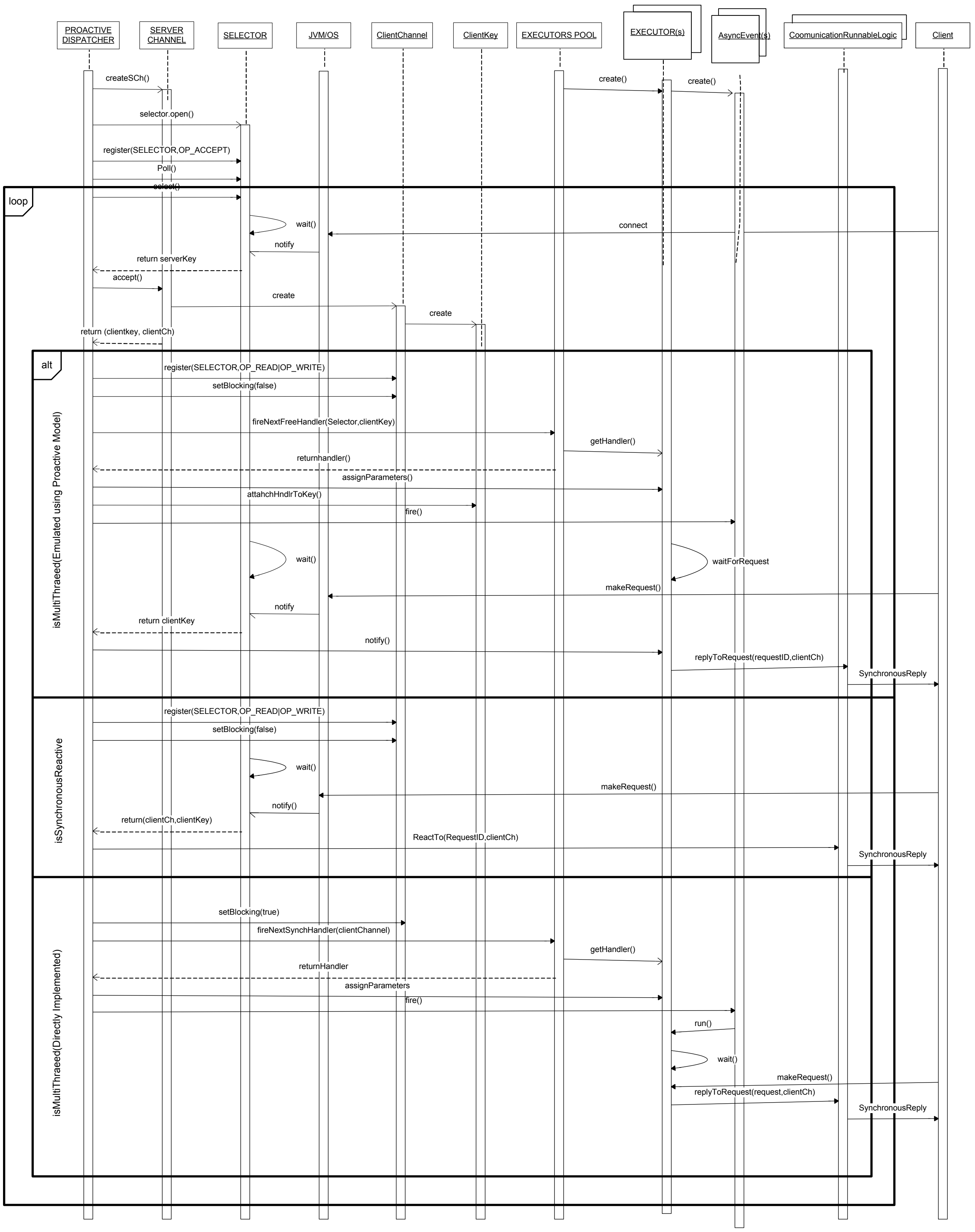

**Figure 2: Sequence Diagram of Different Synchronous Scheduling Configurations of the Server Component**

**(3) Synch Multithreaded Model Execution Phase.**
This model, see figure 2, can be either implemented directly on the blocking channels, or the non-blocking proactor dispatcher model can emulate it. To implement it directly, the socket channel created to handle client requests is configured to be in blocking mode. So, the selector object is not required to be notified of the events occurring on this channel as these events are directly monitored and handled by the executors. Therefore, the proactor dispatcher will work as a connection listener, once a connection is requested; the dispatcher retrieves a reusable executor for this client connection. After the executor starts to run, all the events on the channel are directly execute by it. So, the executor will be responsible of running all the communication logic runnables, i.e. acceptor handler, reader handler, write handler.

To emulate the work of this reactive model, the selector will perform processing, as the server channel will stay configured as non-blocking. The only difference is, instead of creating an executor to handle each event occurs on the client channel, only the first retrieved executor, i.e. the one that runs the acceptor handler, will block waiting for a notification to continue processing the next event handler runnable. Any event occurring on this client channel will be notified to the operating system, then to the selector, and finally to the executor to release the lock and calls the handler of this event.

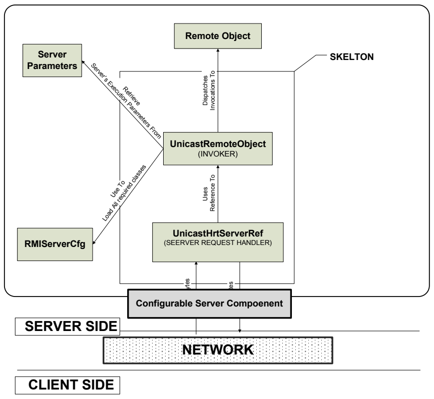
Remote Object
Server Parameters
SKELTON
Retrieve Server's Execution Parameters From
Dispatches Invocations To
UnicastRemoteObject (INVOKER)
Use To Load All required classes
Uses Reference To
RMIServerCfg
UnicastHrtServerRef (SEERVER REQUEST HANDLER)
Configurable Server Compoenent
SERVER SIDE
NETWORK
CLIENT SIDE

**Figure 3: Addition of The Server comp. to the RMI**

## 4.5 AN RMI SERVER OBJECT USING THE CONFIGURABLE SERVER COMPONENT

In this section we discuss how the component model mentioned above can be used within the RMI internal structure to provide predictable RMI with high performance.

**1- Internal RMI structure**

To build a configurable real-time RMI server object, we designed our model as an enhancement of the RTSJ based RMI model presented in [20]. This real-time RMI replaces some of the basic RMI classes by an RTSJ based implementation (see Figure 4) in order to enhance predictability. Furthermore, the model uses a centric approach for loading the execution parameters at both the server and client sides, where the execution parameters are loaded from dedicated objects configured at design time and loaded during the initialization phase.

Figure 4 shows a detailed diagram of the call flow at the server side. This call flow shows that the call handling at the server side is done using a synchronous multithreaded approach of communication with the server as well as the original RMI design. However, in this RTSJ based real-time model, a server thread is listening for incoming calls. Once a call request is received, a session is created using an RTSJ based AEH to handle synchronously the incoming request, by calling the corresponding method from the remote object and returning back the result to the caller.

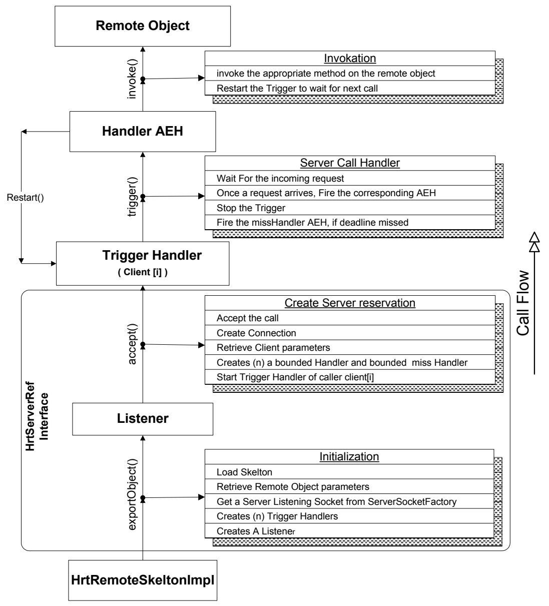
Remote Object
invoke()
Invokation
invoke the appropriate method on the remote object
Restart the Trigger to wait for next call
Handler AEH
Restart()
trigger()
Server Call Handler
Wait For the incoming request
Once a request arrives, Fire the corresponding AEH
Stop the Trigger
Fire the missHandler AEH, if deadline missed
Trigger Handler ( Client [i] )
Call Flow
accept()
Create Server reservation
Accept the call
Create Connection
Retrieve Client parameters
Creates (n) a bounded Handler and bounded miss Handler
Start Trigger Handler of caller client[i]
HrtServerRef Interface
Listener
exportObject()
Initialization
Load Skelton
Retrieve Remote Object parameters
Get a Server Listening Socket from ServerSocketFactory
Creates (n) Trigger Handlers
Creates A Listener
HrtRemoteSkeltonImpl

**Figure 4 Call Flow within the real-time multi-threaded RMI**

To use our proposed server model, this component has to be inserted as a back–end to the skeleton of the server side implementation of the RMI, see Figure 3, to be able to handle the incoming calls. Therefore, to be integrated within the RMI implementation proposed in [20], the component's *Request Handler Logic Runnable* has to provide the protocol implementation of handling the RMI calls. Hence, an encapsulated method class that implements this protocol can be used to configure the component. As defined for the RMI's request handling protocol, the server side is doing the following operations in sequence:

(a) **Handshaking with the server:** This consists of a set of a predefined sequence of exchanged messages holding constant codes and connection information in order to correctly establish the RMI connection.

(b) **Decode the requested method.**

(c) **Create Data Input Stream Object.** To be able to *synchronously* read the incoming data from the client.

(d) **Create dummy output streams objects.** This is responsible for serializing the returned value, if any, to be transmitted back to the client.

(e) **Forward the received call to the remote object to be ececuted.**

(f) **Wait for the result**

(g) **Return the result back to the client through the dummy objects created in (c).**

Hence, the encapsulated method should implement all these functions in a way that is compatible with the configuration assigned to the server component. Since the original model is based mainly on the synchronous multithreaded model, then there will be no critical changes in the above operations when the component is configured to run as multi-threaded server. In contrast, configuring the server component to use proactive method will require important changes to the above operations. Basically, as all the above operations use blocking methods for communication with the client, the first important change is to change all these operations to run asynchronously instead of synchronously. This can be done through running asynchronous calls over the client channel and registered with the selector.

## 5. CONCLUSION

The current Real-time Specification of Java has focused mainly on the scheduling and memory management models for Java non-distributed applications. It is silent about the requirements of the real-time support of distribution and remoting mechanisms and how they can be integrated with the scheduling and memory models of RTSJ. Most of the current research in this direction is targeted at integrating RTSJ within the current RMI structure. However, the current RMI structure is dependent mainly on using a single model of remote server object, the multi-threaded synchronous server model. Although this model has been widely used in the Java language, it is not necessarily the most efficient solution for many distributed real-time systems. This is because this model is limited to the physical resources of the server, and the lack of support of remote asynchronous calls that is a basic requirement of many such systems. Hence, in this paper we provided an initial proposal of a general configurable server component that provides three different server models, proactive asynchronous, reactive synchronous, and the common multithreaded synchronous model.

The aim of integrating these three models in a single model is to provide a high level of abstraction to the developer, as each of these three models has its own range of applications and requirements. For example, the proactive model is the most efficient and scalable one but it requires asynchrony support from the underlying virtual machine and operating system. In contrast, the reactive synchronous model is more efficient in systems with no multi-threading support. On the other hand, the multi-threaded model offers a vey simple approach for systems with no requirements of scalability.

Integrating the proposed server component with RMI, will have all the above advantages of the component to be inherited in the RMI model itself, which gives flexibility to the developer in developing his remote applications. Hence, we have provided a RTSJ based framework of integrating the proposed component with the RMI implementation, with a concentration on the integration of the proactive model of the component, and how this would require a set of modifications in the RMI implementation to adapt to the asynchrony nature supported in this model.

In the future we aim to analyze the component predictability in order to enhance its operation. Moreover, as we provided a framework for the server-side, we aim to create another configurable component to support different model of client-side remoting calls. This client-side component is assumed to support the asynchronous calls models, e.g. Poll Objects, Result call back, as well as the synchronous model. This model is assumed to be integrated with the RMI client side implementation to offer flexibility for the user in making synchronous or asynchronous calls.

## Acknowledgement

We would like to thank the authors of [20] who provided us an open source of their implementation of HRT RMI, which was very helpful during our research.

## REFERENCES

[1] The JSR-001, “*The real-time specification for Java*”, available online at http://jcp.org/en/jsr/detail?id=1

[2] The JSR-050, “*The distributed real-time specification for Java*”, available online at http://jcp.org/en/jsr/detail?id=50

[3] A. Wellings, R. Clark, and D. Jensen, D. Wells, “*Towards A Framework for Integrating the Real-Time Specification for Java and Java's Remote Method Invocation*”, Proceedings of the Work-In-Progress Session, 22nd IEEE Real-Time Systems Symposium, 2001

[4] A. Wellings, R. K. Clark, E. D. Jensen, and D. Wells. “*A framework for integrating the Real-Time Specification for Java and Java’s remote method invocation*”. In Proc. of the 5th IEEE International Symposium on Object Oriented Real-Time Distributed Computing, April 2002.

[5] 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.

[6] Borg A. and Wellings A. (2003), “*A Real-Time RMI Framework for the RTSJ*”, Proc. of the 15th Euromicro Conference on Real-Time Systems (ECRTS’03), Porto, Portugal, pp 238-248.

[7] S. Rho, “*A distributed hard real-time Java system for high mobility components*”, Ph.D. (Texas A&M University, December 2004)

[8] B. Choi, S, Rho, and R. Bettati, "*Dynamic Resource Discovery for Applications Survivability in Distributed Real-Time Systems.*" Proceedings of the 11th IEEE International Workshop on Parallel and Distributed Systems, pp. 122-129, Nice, France, Apr. 22-23, 2003.

[9] D. Tejera *et al*, “*Two Alternative RMI Models for Real-Time Distributed Applications*”. ISORC 2005: 390-397.

[10] J. Kwon, *et al*, ”*Ravenscar-java: a high-integrity profile for real-time java”*: Research articles. Concurr. Comput. - Pract. Exper. 17(5-6), 681–713, (2005).

[11] D. Tejera, *et al*: “*Predictable Serialization in Java*.” ISORC 2007: 102-109.

[12] R. Clark *et al, “An Architectural Overview of the Alpha Real-Time Distributed Kernel*”. In Proceedings of the USENIX Workshop on Microkernels and Other Kernel Architectures, 1992.

[13] M. Voelter, *et al, “Remoting Patterns Foundations of Enterprise, Internet, and Real-time Distributed Object Middlware”,* Wiley Series in Software Design Patterns ISBN: 0470856629, Wiley and Sons 2004.

[14] D. C. Schmidt, *et al*. “*Leader-Followers: A Design Pattern for Efficient Multi-threaded Event Demultiplexing and Dispatching”*. In Proceedings of the 6th Pattern Languages of Programming Conference, Monticello, Aug. 2000.

[15] D. C. Schmidt, “*Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching*”, in Pattern Languages of Program Design (J. O. Coplien and D. C. Schmidt, eds.), pp. 529–545, Reading, MA: Addison-Wesley, 1995.

[16] I. Pyarali, *et al*, “*Proactor – An Architectural Pattern for Demultiplexing and Dispatching Handlers for Asynchronous Events*” in Pattern Languages of Program Design (B. Foote, N. Harrison, and H. Rohnert, eds.), Reading, MA: Addison-Wesley, 1999.

[17] The JSR-051, “*The JSR 51: New I/O APIs for the JavaTM Platform*”, available online at http://www.jcp.org/en/jsr/detail?id=51

[18] The JSR-203, “*JSR 203: More New I/O APIs for the JavaTM Platform ("NIO.2")*”, available online at http://www.jcp.org/en/jsr/detail?id=203

[19] Pizlo F, Fox J, Holmes D, Vitek J (2004) “*Real-time Java scoped memory: design patterns and semantics*”. In: Proceedings of the IEEE international symposium on object-oriented real-time distributed computing (ISORC'04), Vienna, Austria, May 2004

[20] Daniel Tejera, Alejandro Alonso, *et al, “RMI-HRT: remote method invocation - hard real time*”. JTRES 2007: 113-120