Design Patterns and Component Framework
for Building

RTSJ-Based

Real-Time Middleware

Mohammed Fathi Alrahmawy

PhD
The University of York
Computer Science Department
YORK, UK

September 2010

Abstract

The middleware is a software layer located on top of the operating system that uses its facilities, integrates with it, and extends its functionality in order to support the development of effective and reliable distributed systems; however, the architectures of most of the conventional middleware solutions, e.g. Java RMI, do not either offer the predictability required to support the real-time behaviour in these systems, or the reconfigurablity required for these middleware solutions to be applicable in a wide range of distributed systems.

Java has a great support for building distributed systems; however, due to its unpredictability, Java does not support building distributed real-time systems and middleware solutions. Hence, this thesis argues that the RTSJ can be used to build reusable and reconfigurable software components and design patterns that have high levels of predictability and reliability. These proposed real-time components and design patterns can be used for building real-time middleware solutions in Java. Therefore, the RTSJ has to be used as a base for modifying the existing reconfigurable and reusable software patterns and components for distribution models. It could also be used to create a new set of these patterns and components, in order to support the real-time behaviour and the predictability in the real-time middleware.

The key contributions in this thesis include the presentation of a component framework design model for building RTSJ-based middleware and distributed systems; this framework focuses mainly on the memory model, the communication model of building the components and it also provides the management mechanism of these components, which uses a set of design patterns that integrates with these models. This includes a memory model for the RTSJ components associated with a set of reusability and life-management sub-components that support building real-time components in the RTSJ. Also, we provided a design of a real-time reconfigurable communication component based on the RTSJ that can be used to support predictable low level remote communications in distributed real-time Java applications and we showed how this component could be integrated within the component model as a sub-component, to provide communication services within the component model. Also, we presented our own model of integrating both the framework and the component model within the RMI architecture to provide a reconfigurable real-time Java RMI middleware based on the RTSJ.

Table of Contents

List of Figures

Acknowledgements

In the Name of Allah, the Most Gracious, the Most Merciful

Praise be to Allah, the Lord of the worlds, for His favour to me in completing this thesis, praying Him to accept this work from me as a service for His sake, and for the benefit of mankind, and I thank Him for supporting, helping, and guiding me, not just in completing this thesis, but in my entire life from birth to death, and I ask Him for His mercy and forgiveness, and I pray to Him to guide me to the Straight Path in this life and in the Hereafter.

There are many people I would like to thank who have contributed to this thesis in various ways.

Firstly, I would like to thank deeply my parents, who helped, guided, and supported me all my life. Deep thanks especially for my mother, for the encouragement she provided and the sacrifices she made to help me and my sister after my father passed away, which without, I would not be able to achieve my goals and to overcome the difficult times during the PhD journey.

I would like to thank my family, my wife Shaymaa and my two little daughters; Mennat-Allah and Toqa, for their patience, sacrifices and support. I also thank my sister Ghada and her children; Yara, and Yousef, for encouraging and supporting me throughout my PhD.

I would like to thank my supervisor, Prof. Andy Wellings, who has guided and supported me in various aspects through the entire journey in completion of my thesis work.

I would like to thank the Egyptian Government who provided the funding and the support for my work.

I would like to thank all my friends and colleagues in the University of York, particularly the members of the Real-time systems group, who provided me with the help, the advice, and the support during my studies, which made this entire journey much more enjoyable.

At last, but not the least, I would like to thank all my friends, both in my country Egypt, and in the city of York, for their help and support especially during the difficult times during the PhD journey.

Declaration

This thesis has not previously been accepted in substance for any degree, and it is not concurrently submitted in candidature for any degree other than degree of Doctor of Philosophy of the University of York. This thesis is the result of my own investigations, except where otherwise stated. Other sources are acknowledged by explicit references.
Some of the material presented in this thesis has previously been published as follows:

1- Chapter 2, 3, 4 are extension to the material previously published as a technical report entitled “Real-time Middleware” (Computer Science Department, University of York, MPhil, DPhil “Qualification Dissertation, 2006).
2- Chapter 5 is an extension on the work presented in (Alrahmawy and Wellings 2007; Alrahmawy and Wellings 2009).
3- Chapter 6 is based on the work published in (Alrahmawy and Wellings 2009)

All of these publications were written by me with advice from my supervisor, Prof Andy Wellings.

I hereby give consent for my thesis, if accepted, to be made available for photocopying and for inter-library loan, and for the title and summary to be made available to outside organisations.

Signed ……. . . . . (candidate)
Date . . . . . . ……. . . . .

Chapter 1

Introduction

The need for distributed applications is of major concern, and this need is increasing rapidly nowadays and will continue to increase in the next generations. In a wide range of these distributed applications there is an increasing demand for real-time support. At the moment, over 99% of all the microprocessors are used within networked embedded systems (Burns and Wellings 2001) that control, in real time; physical, chemical, biological, or defense processes and devices. This has resulted in a new set of distributed applications known as distributed real time embedded systems (DRE) that need strong support of real-time behavior. Examples of such systems include (Krishna, Schmidt et al. 2003).

1. Telecommunications networks (e.g. wireless phone services)
2. Telemedicine (e.g. robotic surgery)
3. Process automation (e.g. hot rolling mills)
4. Multimedia streaming (e.g. web broadcasting)
5. Avionic systems (e.g. flight guidance and control systems)
6. Defense applications (e.g. total ship computing environments)

The complexity of distributed systems complicates their development and testing, as many of the current languages and tools used for building such systems have low levels of abstraction. Also, the use of highly specialized technologies can make it hard to adapt the software to meet new functional or QoS requirements, hardware/software technology innovation, or emerging market opportunity.

One of the most successful strategies used to simplify and speedup distributed software development is the use of middleware solutions. However, the process of building and designing middleware solutions itself is not an easy task, as different distributed systems have different requirements. Hence, it is difficult to build a single middleware that can be used in all distributed systems. So, in order to ease the building of reliable and efficient middleware solutions and distributed systems, there is a need for defining and constructing tested and generalized reconfigurable software constructs that can be used in building such systems.

The development of efficient middleware architectures that ease, enhance, and speed up the building of distributed systems in general and in particular for developing distributed real time systems is a common challenge. Reusable software patterns and components technologies are two main technologies that integrate together in order to provide reusable software constructs for building software systems. Therefore, these technologies have been widely used for developing many conventional middleware solutions.

However, the conventional middleware solutions have been found not to be suitable to build most of the distributed real-time and embedded systems. One of the main reasons of this, is that the conventional technologies of software design patterns, and components off the shelf (COTS) used in building such middleware solutions have been found unsuitable for use in DRE systems due to either being (Schmidt 2002):

- **Flexible and Standard;** but incapable of guaranteeing stringent QoS demands, which restricts assurability.
- **Partially QoS-Enabled;** but inflexible and non-standard, which restricts adaptability and affordability.

An additional important challenge that faces the development of middleware solutions is the complexity of building reliable communication mechanisms and paradigms over the networks. Hence, many programming languages provide integrated communication and networking mechanisms to ease the development of distributed software systems (for example, Java and Ada).

The Java language is one of the best programming languages in providing integrated communication and networking mechanisms. There are many distribution middleware solutions built using Java. RMI is the basic middleware provided in the Java language, and many other distribution middleware solutions have been implemented over it. However, in addition to the unpredictability of the Java language and the lack of support of QoS guarantees or the end-to-end timeliness, the Java middleware solutions have not been widely used to build distributed real-time systems. So there is a high demand for enhancing the Java middleware technologies, particularly RMI, to push it for use in distributed real-time systems. We aim in this research to see how the use of the Real-Time Specification of Java can help to build real-time reconfigurable middleware in Java.

1.1 Overview of Distributed Real-Time Systems

The revolution in the communications and software technologies caused distributed real time systems to grow in size and complexity. The development of distributed embedded real-time systems faces key technical challenges. One of these major challenges is satisfying multiple QoS requirements in real-time. Examples of QoS requirements include (Wang, Schmidt et al. 2003):

- **Real-time requirements**; that guarantee the end-to-end timeliness, low latency and bounded jitter.
- **High availability requirements**; such as fault propagation/recovery and load balancing across distribution boundaries.
- **Physical requirements**; such as limited weight, power consumption, and memory foot print.

To ensure that the DRE systems can achieve their QoS requirements, various types of QoS provisioning must be performed to allocate and manage system computing and communication resources end-to-end. This can be performed in the following ways (Wang, Parameswaran et al. 2001):

- **Statically**; by ensuring that adequate resources required to support a particular degree of QoS is pre-configured into an application. Examples of this are task prioritization and communication bandwidth reservation.
- **Dynamically**; where the resources required are determined and adjusted at run time. Examples of dynamic QoS provisioning include; run time reallocation and re-prioritization to handle burst CPU load, and competing network traffic demands.

Due their characteristics, reliability is of a major concern in real-time system. Reliability of a system is a measure of success of how the behaviour of this system meets its required specification; hence, system specification should be complete, consistent, comprehensible, and unambiguous; otherwise, the behavior of the system deviates from that which is specified for it, this deviation is called a *failure*. Failures result from unexpected problems internal to the system, which eventually manifest themselves in the system's external behavior. These problems are called *errors*, and their mechanical or algorithmic causes are called *faults*. In general, any system is usually composed of components, each of these components may be considered itself as a smaller system; hence, a failure in any of those smaller systems may lead to a fault in another, which results in an error and potential failure of that system. This in turn introduces a fault into any surrounding system and so on. (Burns and Wellings 2001).

Real-time middleware is categorized as a subset of distributed real-time systems. This means real-time middleware has all the characteristics of both real-time systems and distributed systems with additional properties that specializes it. In this section we aim to discuss the basic definitions and properties of both real time systems and distributed real time systems in order understand the characteristics of real-time middleware, then we will show some middleware definitions that identify its specific properties.

1.1.1 Definitions of Real-Time Systems

There are several definitions of real time systems amongst different groups, such as vendors, software developers, practitioners, academics, researchers and so forth. Some of these definitions are given below.

The Oxford dictionary of computing offers the definition as:

> *"It is a system in which the time at which the output is produced is significant. This is usually because the input corresponds to some movement in the physical world, and the output has to relate to that same movement. The lag (delay) from the input time to output time must be sufficiently small for acceptable timeliness".*

The above definition covers different types of systems, from workstations running under UNIX operating systems, where the user expects to receive a response within a few seconds, to aircraft engine control systems which must respond within a specified time. Failure to do so could cause the loss of control and possibility loss of passengers lives (Palue 2002).

A second definition was presented in the *Journal of Systems and Control Engineering,* and it defines real-time systems as follows (Cooling 1991):

> *"Real-Time Systems are those which must produce correct responses within a definite time limit. Should computer responses exceed these time bounds then performance degradation and/or malfunctions results."*

From a software developer point of view, the following is an alternative definition to the above, presented in (Palue 2002):

> *"Real-Time Systems read inputs from the plant (a physical system to be computer controlled, e.g. robot, supermarket automated*

> *entrance sliding door, factory automation process, digital camera and so forth) and sends control signals to the plant at times determined by plant operational considerations - not at times limited by the capabilities of the computer systems".*

From all the above, we can see that all the definitions of real-time systems are referring to the systems that have execution time constraints, which have to be satisfied otherwise the system performance would degrade and it might fail to provide its predicted functionality required from it.

1.1.2 Classification of Real-Time Systems

As concluded from their definition, real time systems are systems in which time plays a critical role in its functionality. A distinction can be made among those systems which will suffer a critical failure if time constraints are violated (hard or immediate real-time), and those which will not (soft real-time). It is important to note that hard versus soft real-time does not necessarily relate to the length of time available (Laplante 2004). A machine may overheat if a processor does not turn on cooling within 15 minutes (hard real-time). On the other hand, a network interface card may lose buffered data if it is not read within a fraction of a second, but the data can be resent over the network if needed, without affecting a critical operation, perhaps without a delay noticeable to the user. According to this, the following definitions were presented for the different types of the real-time systems:

- **Soft Real time System**: This type of systems was defined in (Laplante 2004) as;

> *"A soft real-time system is one in which performance is degraded but not destroyed by failure to meet response-time constraints"*

This definition means that missing even many deadlines will not lead to catastrophic failure, only degraded performance. Hence, a soft deadline will often have a few characteristics, which describe the deadline (Newcombe and Seraj 2002) including the deadline itself, the upper bound on the probability of missing the deadline, and an upper bound on the lateness of the delivery.

- **Hard Real-Time System:** This type of the real-time systems has a high level of timing constraints as seen in the following definitions (Laplante 2004);

> *"A hard real-time system is one in which failure to meet a single deadline may lead to complete and catastrophic system failure"*

For instance, missing the deadline to launch the missile within a specified time after pressing the button can cause the target to be missed, which will result in catastrophe.

- Firm Real time System: Several definitions of firm real-time systems exist. In (Laplante 2004) it is defined as;

> *"A firm real-time system is one in which a few missed deadlines will not lead to total failure, but missing more than a few may lead to complete and catastrophic system failure". For instance missing critical navigation deadlines causes the robot to veer hopelessly out of control and damage crops"*

A slightly different definition is found in (Burns and Wellings 2001), where it is defined as:

> *"A deadline that can be missed occasionally, but in which there is no benefit from late delivery, is called firm."*

A third definition is found in (Newcombe and Seraj 2002), defines it as:

> *"The firm real time is a variation of soft real time system. The firm real time system will recover from a missed deadline but once the deadline is missed, the activity is stopped. It is important to note that the obvious objective is to meet these deadlines but failure to do so is not catastrophic"*

In conclusion, according to the above definitions of real-time systems types, we can represent the real time systems as shown in the diagram in Figure 1-1, which shows the relation between the correctness-value of the system against the number of deadlines.

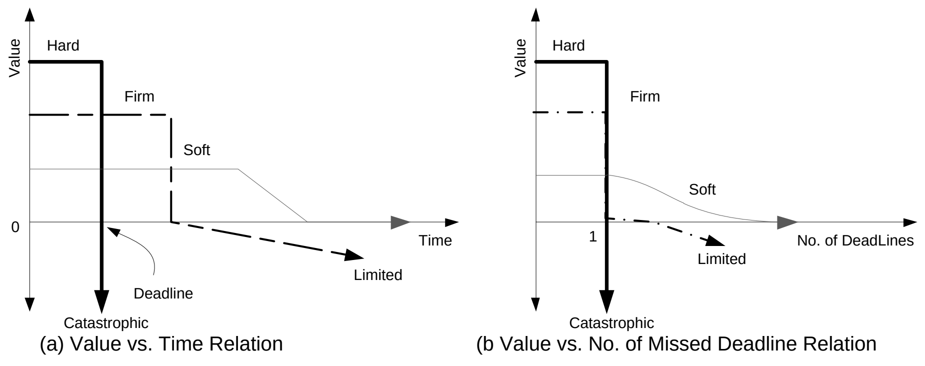


Figure 1-1 Types of Real-time Systems

1.1.3 Definition of Distributed Real-Time Systems

Distributed real time systems are a special category of real time systems that apply the real-time constraints defined for real time systems to distributed systems. So, a definition for distributed system is needed first before providing a definition of such systems. As it was in the case for the real-time systems, there are many definitions of the distributed systems that say similar things; some of these definitions are given below:

In (Burns and Wellings 2001) a distributed system is defined to be:

> *"It is a system of multiple autonomous processing elements cooperating in a common purpose or to achieve a common goal."*

This definition is a wide definition of a distributed system, without descending to details of physical dispersion, means of communication and so on.

Another definition of distributed systems is found in (FOLDOC 1994) as:

> *"It is a collection of (probably heterogeneous) automata whose distribution is transparent to the user so that the system appears as one local machine. This is in contrast to a network, where the user is aware that there are several machines, and their location, storage replication, load balancing and functionality is not transparent. Distributed systems usually use some kind of client-server organization"*

A third detailed definition of distributed systems is found in (Microsoft 2002), and defines the distributed systems as:

> *"A non-centralized network consisting of numerous computers that can communicate with one another and that appear to users as parts of a single, large, accessible 'general storehouse' of shared hardware, software, and data. A distributed system is conceptually the opposite of a centralized, or monolithic, system in which clients connect to a single central computer, such as a mainframe."*

As they are concerning distributed systems in general, the above definitions do not say anything related to real time. However, we can consider distributed real-time systems as a system that has a combined definition of both real-time systems and distributed systems. For example, one of the more explanatory and specific definition co-related to real-time systems defines distributed real time system as(Urbano 2002):

> *"A distributed real-time system is an integrated system composed of a set of dedicated hardware that monitors real-world processes; acts and reacts on events respecting time requirements. The elements of such system are inherently concurrent, and the need of synchronization arises when co-operation is required. The co-ordination of action between the elements is achieved using a shared resource such as a communication channel in which the elements exchange data and messages. "*

1.2 Overview of Real-time Middleware

As stated before, middleware technology has a set of characteristics and properties that are specific to it and make it different from other types of distributed systems; here we have discussed these characteristics, then we will consider how the different types of the middleware can be classified from different points of views.

1.2.1 Definition of Middleware

Many interesting definitions of middleware exist, all centered on sets of tools and data that help applications use networked resources and services. This breadth of meaning is reflected in the following working definition provided by :

> *"Middleware is the intersection of the stuff that network engineers don't want to do with the stuff that application developers don't want to do"*

In (The Computing dictionary 2010) middleware is defined from a functional point of view as:

> *"It is software that functions as a conversion or translation layer. It is also a consolidator and integrator"*

Gartner Group (commercial software provider) (Gartner Inc 2001) defines middleware as:

> *"It is run time system software that directly enables application level interactions among programs in a distributed computing environment"*

In (Schantz and Schmidt 2001) it as defined as follows:

> *"Middleware is reusable software that resides between applications and the underlying operating system, network protocol stacks, and hardware"*

1.2.2 Middleware Characteristics

The purpose of using middleware is to isolate the application from the platform specific differences, both hardware and software, and provide facilities to hide the undesirable aspects of distribution. These are often referred to as distribution transparency mechanisms that can be classified into the following different aspects (Macmillan 1995):

- **Location Transparency:** masking the physical locations from services.
- **Access Transparency:** masking differences in representation and operation of the invocation mechanisms.
- **Concurrency Transparency:** masking overlapped execution.
- **Replication Transparency**: masking redundancy of the resources.
- **Failure Transparency:** masking recovery of services after failure.
- **Resource Transparency:** masking changes in the representation of a service and resources used to support it.
- **Migration Transparency:** masking movement of service from one application to another.
- **Federation Transparency:** masking administrative and technology boundaries.

Properly developed and deployed middleware can reduce the task of developing distributed applications and systems by helping to (Wang, Schmidt et al. 2003):

1- Provide a set of capabilities closer to the application design level abstractions to simplify the development of distributed applications.
2- Manage system resources by using higher-levels of abstractions.
3- Avoid the use of the low level, tedious and error-prone platform details.
4- Reduce system-lifecycle costs by building trusted reusable software patterns.
5- Provide a wide array of the ready to use services for developers.
6- Ease the integration and interoperability of software over diverse heterogeneous and separated environments.
7- Provides industry-wide standards for the higher levels abstraction of portable software.

1.2.3 Middleware Classifications

Several standardization efforts are ongoing in several areas of middleware. These efforts have resulted in different classifications of middleware. Some of these classifications are discussed below.

A- Architectural-Based Classification

A classical classification of middleware solutions classifies them according to their design and architectural elements used to build them as follows (Duran-Limon, Blair et al. 2004):

- **Remote Procedure Calls Middleware (RPCM)**: By allowing procedures in heterogeneous distributed platform to be called as if they were local. For example many operating systems support the Open Group's DCE Standard.
- **Transaction-oriented Middleware (TOM)**: Aims to interconnect heterogeneous database systems, offering high performance, availability and ensure data integrity database systems.
- **Message-Oriented Middleware (MOM):** Provides asynchronous rather than synchronous interactions.
- **Object-Oriented Middleware (OOM):** Supports the remote invocation of object methods. CORBA, Java RMI, DOCM, and .NET are important OOM platforms.
- **Component Oriented Middleware (COM):** Enables reusable services to be composed, configured and installed to create distributed applications rapidly and robustly. Examples of these technologies include the CORBA component model (CCM) and the Enterprise Java Beans (EJB).

B- Heterogeneity-Based Classification

A broader approach for classifying middleware was defined in (**Medvidovic** 2003) according to the type of heterogeneity as follows:

- **Platform heterogeneity**. Middleware can allow communication among components running on different platforms. For example, many CORBA ORBs have compatible implementations on various flavors of UNIX, Windows, etc. Java based middleware like RMI (Plášil and Stal 1998) also allow this, but depends on the portability of the underlying virtual machine.
- **Language heterogeneity**. Middleware can allow communication among components written in different programming languages. Microsoft's COM, for example, allows communication among components written in Visual Basic, C++, and

other Microsoft languages. In contrast, RMI middleware allows communication among components written in Java only.

- **Connectivity_heterogeneity**. Middleware can allow the ability to store-and-forward information. This middleware is useful for connecting components with unreliable network connectivity to other components or components that are nomadic. For instance, QoS-enabled middleware for media delivery can download sample multimedia stream that is viewable by someone with 56kbit connection.

C- Classification According to the Non-functional Requirements

A middleware taxonomy with respect to non-functional requirements can be found in (FP6IPRUNES 2005); it depends on the fact that middleware itself is a distributed system that needs to meet the requirements of any type of distributed system. So any middleware can be evaluated and classified according to whether, and in what degree, it can meet each individual requirement. These requirements are (Huston and Schmidt 2001; FP6IPRUNES 2005):

- **Heterogeneity;** The capability of working in different programming languages, running on different operating systems and executing on different hardware platforms.
- **Openness;** The capability to extend and modify the functionality of the middleware.
- **Scalability;** The ability of the system to accommodate a higher load at some time in the future.
- **Failure handling;** the ability to recover from faults without halting the system.
- **Security**; Mechanisms such as authentication, authorization, and accounting functions may be an important part of the middleware in order to intelligently control to system resources, enforcing policies, etc.
- **Performance;** It can constitute a requirement of the middleware in various situations as in QoS and real-time systems.
- **Adaptability**. The presence of adaption mechanisms within middleware may be needed to cope with changes in the applications' and users' requirements.
- **Feasibility**. Constraints of available resources may limit the feasibility of performing certain tasks or offering certain services in a given environment.

D- Layered-Based Classification

Just as a networking protocol stack can be decomposed into multiple layers, a specific classification that is specific to the Object Oriented Middleware was presented in (Schmidt 2002), and decomposes middleware into multiple layers, shown in Figure 1-2.

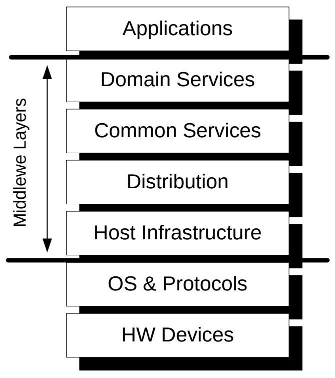


Figure 1-2 Middleware layers

According to this classification, the common hierarchy of object-oriented middleware includes the layers described below:

- **Host Infrastructure Middleware;** encapsulates and enhances native OS communication and concurrency mechanisms to create portable and reusable network programming components, such as monitor objects and active objects. These components help eliminate many tedious, error-prone and non-portable aspects of developing and maintaining networked applications; via low level OS programming (e.g. Java virtual machines).
- **Distribution Middleware;** defines higher-level distributed programming models whose reusable APIs and mechanisms automate and extend the native OS network programming capabilities encapsulated by host infrastructure middleware. It enables developers to program distributed applications much like standalone applications, i.e.; by enabling the invocation of operations from target objects regardless of location, OS platform and communication protocols (e.g. CORBA and Java RMI).
- **Common Middleware Services;** extends distribution middleware by defining higher level domain-independent reusable components that allow application developers to concentrate on programming application logic, without the need to write the plumbing code needed to develop distributed applications by using lower level middleware features directly. It focuses on allocating, scheduling, and coordinating various end-to-end resources throughout the distributed system using a component programming and scripting model (e.g. CORBA Services, CORBA component Model).
- **Domain Specific Middleware Services;** these services are tailored to the requirements of a particular DRE system domains, such as avionics mission computing, telecommunications, e-commerce, health care, etc. Boeing Bold Stroke architecture for mission computing avionics capabilities is an example of such middleware services (Sharp 1998).

1.3 Motivations and Research Scope

The Java programming language is one of those languages that support the mapping of operating systems facilities for communication and networking facilities, e.g. sockets, into very efficient and highly abstracted libraries, which are easy to use by developers. This support, along with Java's strong semantics and object-oriented programming model, and its support for building reusable components, has resulted in Java being one of the first choices for distributed software designers and developers when building highly efficient non-real-time middleware for distributed systems.

Java not only provides packages that abstract the low level communication operations offered by operating systems, but also, it comes with a remote communication middleware solution, the Java RMI. Java RMI is a middleware that enable the invocation of methods defined in remote objects that exist on remote nodes, as if they are invocations to local methods, i.e.; it hides the complexity of using the low level operations such as the initiation of the connection, the transfer of objects into bytes to be sent over the network, the locating of the remote method, the passing of arguments to it, executing it, and the return of the result from the server object to the client as bytes over the network, and then it rebuilds the returned object and delivers it to the client.

The importance of Java RMI is not just because it has the basic middleware that is presented in Java, but also because it represents the base layer of other advanced middleware solutions that have higher level of abstractions such as Jini. So, developing real-time RMI is a key element for building advanced real-time middleware solutions in Java.

However, the conventional Java RMI middleware implementation is lacking a lot of the important features required for real-time systems. The most essential reasons for this are:

1- Java RMI is built using the Java language, so it inherits the unpredictability of the Java language, which was the reason that Java has not found the same success in building real-time systems; this unpredictability is due to the lack of support of predictable memory and scheduling models.
2- The software patterns used for the implementation of the Java RMI both at the server side and the client side use only a single model of communication, blocking communication, which is not suitable for many distributed real-time systems.

3- Java RMI has no built-in mechanism to guarantee the required end-to-end timelines of the remote call execution.

Much of research has been attempted to overcome the unpredictability of the Java language. This research has resulted in the release of the Real-Time Specification for Java (RTSJ) (G. Bollella, B. Brosgol et al. 2006), that has been proposed to provide the required extensions necessary to be integrated with the Java platform to provide more predictability.

The RTSJ is an extension to Java that aims to solve the unpredictability problems of Java and to support the real-time concepts and requirements directly in the language itself. The RTSJ provides a predictable memory model that uses scoped memory areas to avoid the unpredictability due to the garbage collector. Also, RTSJ defines a scheduling model that provides an integrated real-time scheduler and predictable schedulable objects.

However, the RTSJ has not targeted distributed systems, as it focused only on centralized systems. Hence, using it to build real-time RMI middleware faces additional challenges to the above challenges, not only due to the real-time constraints, but also due to its new memory model and scheduling model. This is because:

1- The currently used patterns and components in the current Java-based middleware solutions require modifications to the existing architectural patterns and models used to build them, or even the invention of new ones, in order to be able to provide the patterns necessary to build reusable software components.
2- Building reusable software components in RTSJ for real-time systems using the current component-based systems strategies is a complicated task, and it is not easy to enforce the use of the RTSJ rules into them, especially when components integrate together.
3- Most of current communication and networking technologies have been designed and built without, or with a limited, consideration of supporting real-time behavior. However, RTSJ is silent on providing communication mechanisms suitable for distributed real-time systems.
4- The diversity of the types, platforms and requirements of distributed real-time systems, requires flexible and reconfigurable architectures of both the middleware and the software components that can be easily configured by the developer in order to be used according to the requirements of the target distributed system.

Therefore, in order to integrate Java RMI with the RTSJ, researchers in (A. Wellings April 2002) proposed three levels of integration (the three levels are discussed in chapter 3). Also, in (Clark, Jensen et al. 2002) the users proposed an extension of the Java RMI model that adopts the distributed thread model as a base for the distributed real-time specification of Java, DRTSJ. There is still much work needed to cover all the aspects needed to support a complete model for the real-time RMI.

1.4 Thesis Goals, Hypothesis and Contribution

The Java-based middleware technologies, particularly Java RMI, have been shown to be appropriate for building distributed systems. However, due to both the unpredictability of the Java language and the unpredictability of the software patterns used in building them, they are not suitable for building distributed real-time systems. In addition to the unpredictability, many of the software patterns used for building Java middleware solutions are not flexible enough to be used in the wide range of distributed real-time systems that have different architectures, requirements and functionalities. So, building distributed real-time systems in Java requires not just a real-time support from the language itself, but it needs a set of reusable and reconfigurable software models and constructs for distribution. These software models must have high degree of predictability in order to ease the development of these real-time systems.

The presentation of the RTSJ is a promising step toward taking the Java language to the area of developing real-time systems, since it provides predictable scheduling and memory management models that are integrated within the language itself. So, our first goal in this thesis is to determine the extent to which new memory and scheduling models in the RTSJ can be used to build an RTSJ-based real-time middleware with high levels of abstraction, and how the RTSJ features and the real-time requirements of the middleware can affect the design phases of building it.

Since the RTSJ memory and scheduling models have a set of constraints and rules, the current design patterns and component models cannot be directly used within the RTSJ-based real-time middleware solutions. Hence, our second goal is to provide reusable software design patterns and simplified component models that support building reusable components models that are compatible with the RTSJ memory models and hides its complexity.

Also, as the communication mechanisms of distributed systems can affect greatly the predictability of the system, one of our goals is to see how the RTSJ can be integrated with efficient communication software patterns to build a reusable and predictable communication software facility that offers different mechanisms of communications, that enable the developer either to use it directly within a distributed real-time system or to use it as the communication layer of a real-time middleware solution of higher abstraction.

Finally, Java RMI is not just the main distribution middleware in Java, but also, it has been used as an underlying layer in many Java-based distribution middleware solutions that have higher levels of abstractions, e.g. Jini. Hence, one of our main goals is to investigate its architecture to find out the sources of the unpredictability in it, and then provide a proposal for the changes and enhancements required of the software patterns used for building it; both at the server side and the client side parts of it, in order to provide a real-time model of it.

In relation to above goals, the thesis is stated in the following:

> *"The Java platform does not provide sufficient support for building real-time middleware for distributed real-time applications in general, and particularly the middleware solutions. However, it is possible to enhance the Java platform's ability to build such real-time middleware solutions, by building a component framework that supports building real-time components, where these components are built using predictable RTSJ-based design patterns. This framework can support the building of simplified and flexible reusable components, which hide the complexity of the RTSJ memory and scheduling models, in addition to supporting configurable and flexible low-level communication services that can integrate with the component model, to implement higher middleware models such as RMI, in order to support building distributed real-time applications"*

In order to support this hypothesis, the thesis provides the following contributions:

In order to support the RTSJ based component-oriented real-time applications, we propose a component framework model that integrates the real-time requirements into the component model, and abstracts the complexity of the RTSJ using carefully selected design patterns. In this framework, we propose a memory model for the internal design of the RTSJ-based components and we associate with it a set of design patterns that integrate with the memory model; including a novel set of patterns for managing the life-time of the memory model, including the *ForkThread*, *Dual*

ForkThread, Pinnable Scoped Memory, the Runnable Stack pattern and patterns for easy memory sharing within the component.

Then, we present a model for a configurable real-time communication component based on the RTSJ, the *Real-Time Communicator*, which can be used either as a separate component or as a sub-component within other components. The design of this component adopts the non-blocking mechanisms of communication as its basic model for supporting communication; which is more efficient for real-time systems. The flexible structure of the component enables configuring the component to use it for other models of communications; e.g. blocking synchronous by emulating these models within the component's internal elements. The component supports two modes, a server mode to accept connections and a client mode to initiate connections, while, the handling of the communications events is done internally using a pool of RTSJ schedulable objects to limit the concurrency, and to ensure the predictability.

We provide a new architecture of the Java RMI that evaluates the real-time communication model and the component model by using it at both the server side and the client side. This proposed Java RMI architecture inherits the efficiency and the predictability of the communicator component and can work in several configurations both at the server side for handling the calls, and at the client side for invoking calls.

The model supports using the *FUTURE* objects design pattern to enable the use of the *POLL* object invocation pattern for making non-blocking calls where the RTSJ model of this invocation pattern is integrated within the stub of the remote object at the client side.

We propose an architecture in which the same communication component can be used by several stubs at the same time.

We provide an evaluation of the use of our forked memory model and its associated lifetime management models, e.g. *ForkThread, Dual ForkThread, Pinnable Scoped Memory*, etc. within the stub to specify the constraints of using them.

1.5 Thesis Structure

In order to prove the above thesis hypothesis, the thesis is set out as follows: In chapter 2 and chapter 3 the general and specific approaches for presenting middleware for real time systems are presented. In chapter 2, a hierarchy of middleware paradigms is presented, this hierarchy categorizes the general paradigms of middleware into layers presented from the lowest level to the highest level of abstraction, where for each layer of abstraction we provide the main structural patterns of the middleware technologies lying in this layer associated with the efforts made, within our knowledge, to support the real-time behaviour within these middleware technologies. Then, in chapter 3, we investigate why Java is an efficient language for building distributed systems but not real-time distributed systems; then we provide a brief overview of the architectures of the distributed object paradigm in Java; e.g. Java RMI and CORBA, as they are the most widely used and efficient paradigm for remote communication. Then we investigate in details the reasons that make these middleware technologies not predictable, where our investigation will focus mainly on the problems and requirements in the Java RMI, resulting from the Java and JVM limitations, the RTSJ support, the RMI tools, the RMI programming model and the RMI implementation model. Then, we review the literature of the present research made toward supporting predictability and real-time behaviour in the RMI and RT-CORBA using the RTSJ including the three levels of integrating the Java RMI and RTSJ.

In the first part of chapter 4, we revisit the RTSJ to provide a deeper overview of its new memory and scheduling models in order identify the constraints of using these models and how these constraints have to be considered when designing RTSJ-based systems. Then in the second part of this chapter we introduce both the software design patterns and the components software engineering technologies that enable reusability. The idea of introducing these technologies is that they have proved to be efficient technologies for building guaranteed and reliable middleware solutions. After introducing each of these technologies, we discuss their applicability for use in real-time systems and the research efforts made to provide standardized RTSJ-based design patterns and component models, especially to overcome the constraints provided in the beginning of the chapter.

As the use of the new scoped memory areas for predictable memory management in the RTSJ has a set of constraints, existing software design patterns cannot be mapped directly to the RTSJ, due to the complexity of the internal communication mechanisms. Hence, in chapter 5, we try to hide the complexity of the RTSJ memory model by providing a general component framework that integrates a set of design patterns that enable building RTSJ components, as well as providing pluggable real-time supporting facilities such as thread pooling and remote communication into the component hierarchy.

After building our component framework in chapter 5, we move in chapter 6 to survey the remote communication mechanisms, and their pros and cons for building distributed real-time systems, and from this survey we provide our own novel design and implementation of a reconfigurable low level communication component that adopts the non-blocking mechanisms for processing the networking mechanisms, where the internal design of this component can be reconfigured through a set of configurable properties in order to use it either in a server or a client mode, and to reconfigure it to emulate other communication mechanisms that are usable in a wide range of distributed real-time systems. The handling of the networking communication events is done in this component using a pool of the RTSJ's schedulable objects that can have a limited size to limit the concurrency level within the component, to ensure the predictability. The model is presented in a design that enables using it either as a sub-component within the hierarchy of other components, or as a separate component that integrates with other components.

In chapter 7 we show an analytical evaluation of our communicator component, where we show that the real-time communicator component provides predictable low level of networking communication, which can be used as the communication networking layer of many real-time Java middleware solutions. Also, the proposed memory model of the component and/or its associated design patterns can be used in the building of the Java's real-time middleware solutions. We start the chapter by first analysing the general design patterns used in the general remote middleware, then we show how these design patterns are mapped into the architecture of the RMI-HRT, which is an open source of the Java RMI that support real-time features. We analyse the structure of this middleware, to find the defects that exist in it and then, we propose our own modifications to this middleware which we consider important to avoid the blocking in the original model. Where our main modification is the integration of the communicator component at both the server side and the client side to investigate how it can handle the networking I/O operations. Also, we specify other changes that are required on the RMI protocol to support the use of this component. We then evaluate how the embedding of the communicator component can provide different configurable models of making and handling the remote calls on both the server side and the client side. Then, we discuss the possibility of integrating the component among several stubs concurrently. Finally, we made an overview of the serialization within the model and how we could extend it to support the propagation of the client's temporal parameters.

In chapter 8, we provide additional evaluations that include evaluation of the patterns and subcomponents used in developing our proposed component model, we then evaluate the use of the life control memory management patterns in processing future calls on the client side. Then, we evaluate the support given by the component model to solve the unbounded memory usage of the Java NIO as the basic library used for developing the communicator component.

Finally, in chapter 9, we provide our conclusion of the work done on the thesis, and discuss the possible future work that can extend the work presented in this thesis.

1.6 Summary

In this chapter, we introduced to our research in the real-time middleware, where we provided an overview of the real-time middleware, its characteristics, categories, and the challenges of building it. Then we presented the scope of work of developing real-time middleware in Java using the RTSJ, We presented our hypothesis and the thesis plan to proof the hypothesis. In the next chapter, we provide the structure and paradigms used for building middleware in general and examples of applying them in the real-time domain.

Chapter 2

Structural Paradigms and Technologies *for* Real-time Middleware

The emergence of distributed computing as a dominant computing paradigm is well acknowledged. Along with the increased demand for such applications came the need for tools to facilitate their development. Middleware is a common solution for the development of such applications. In this chapter, a review of distributed systems and middleware paradigms is presented and examples of using these paradigms in developing distributed real time systems, if they exist, are discussed.

Many architectural paradigms and models are commonly used in the development of distributed systems, here we provide a classification of ten of these paradigms and models that are used to build middleware systems and classify them according to their level of abstraction; this classification is an extension of the classification that was presented in (Liu 2001) to classify *some* of these paradigms, see Figure 2-1. We will present ten different patterns starting from the lowest level of abstraction to the highest, where we will present each paradigm with its architectural pattern(s) and discuss the efforts that have been made to apply it in building real-time middleware.

1. The Message Passing Paradigm
2. The Client Server Paradigm
3. The Peer-To-Peer Paradigm
4. The Message System Paradigm

    a- The Point-To-Point Message Model.
    b- The Publish/Subscribe Message Model.

5. The Remote Procedure Call
6. The Distributed Object Paradigm

    a- Remote Method Invocation.

b- Network Service.

c- Object Request Broker (ORB).

d- The Object Space Paradigm.

7. The Application Server Paradigm
8. The Tuple Space Paradigm
9. The Collaborative Application (Groupware) Paradigm

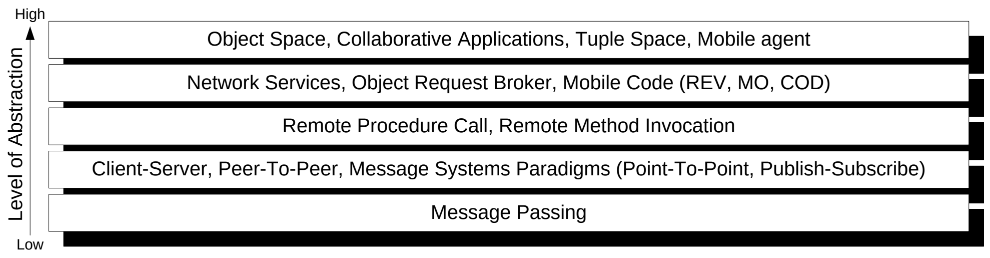

High
Low
Level of Abstraction
Object Space, Collaborative Applications, Tuple Space, Mobile agent
Network Services, Object Request Broker, Mobile Code (REV, MO, COD)
Remote Procedure Call, Remote Method Invocation
Client-Server, Peer-To-Peer, Message Systems Paradigms (Point-To-Point, Publish-Subscribe)
Message Passing

Figure 2-1 Distributed Computing Paradigms and their Levels of Abstraction (BASED ON (Liu 2001))

2.1 Message Passing Paradigm

Message passing is the basic approach for inter-process communication, where data messages are exchanged between two processes, a sender and a receiver. Figure 2-2 illustrates the message passing paradigm where in (a) the process A sends a request message to process B. The message is delivered to process B, which processes the request, and sends a reply message (b) back to process A. In turn, the reply may trigger a further request (c), which leads to a subsequent reply, and so forth. A direct application of this approach is the socket application-programming interface.

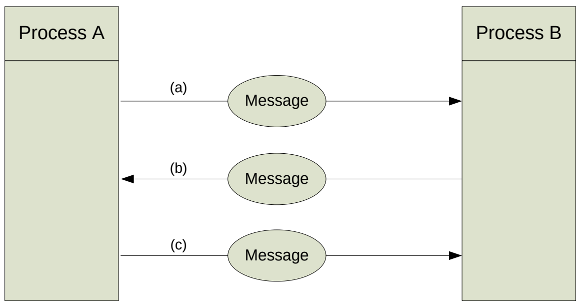


Figure 2-2 Message Passing Paradigm

As the message passing paradigm is at the bottom level of distributed systems paradigms, designers of real-time middleware (based on this paradigm) are usually overly involved with their immediate target platform to make reasonable use of its communication features in order to get the most valuable quality of service for this platform. This makes this middleware non-portable to other platforms (2002; Skjellum, Kanevsky et al. 2004).

To achieve the portability in real-time applications and communication middleware implemented based on the message passing paradigm; researchers at Mississippi State University published *The Real-Time Message Passing Interface (MPI/RT-1.1) Standard* in 2002 (MPI-RT 2002). The design of MPI/RT was based on MPI specification (Message-Passing-Interface-Forum 1994), the widely used standard of message-passing, but provides an application programmer interface that enable *plan in advance, reserve resources, admit sets of channels with timing guarantees*. The standard has been designed using object-oriented design principal and defines bindings for C and C++. However, it does not preclude implementations for other languages.

In MPI/RT, the real time parallel programming world is classified into three categories according to their requirements for QoS as follows (Kanevsky, Skjellum et al. 1997):

- **The Time Driven, Real Time Paradigm**; specifies timing intervals during which messages transfers over channels should take place.
- **The Combined Event-Driven-With Priority Real Time Paradigm;** specifies priority as the real time requirement and events for the start and completion for a channel message transfer, handler execution, or event delivery from a trigger to a receptor. Where transferred messages inherit the channel priority that is specified as the channel QoS. Moreover, any application that requires multiple priority levels can construct multiple “parallel” channels with different priorities to achieve this requirement.
- Time-Driven-With Priority Real Time Paradigm. This paradigm specifies either a priority as the real-time requirement, a timeout as the completion event, or a deadline as both the real-time requirement and the completion event.

The design of MPI/RT adopts the concept of deferred early binding, in which the application is required to set up the communication architecture and its parameters but the actual resource allocation is performed when the application enters its real-time phase. In order to emphasize the deferred early binding approach, MPI/RT defines four phases for the real time application as follows (MPI-RT 2002):

- **Initialization Phase;** initializing both the MPI/RT library and the various processes that will participate in the real time communication.
- **Non-Real-Time Phase;** starts once the initialization phase succeeds in order to create all the required MPI/RT objects with their requested QoS, without allocating system resources for these objects.

- **Real-Time Phase;** starts by allocating system resources and performing global admission testing within the MPI/RT implementation to guarantee the predictability of communication and other MPI/RT operations. The admission test matches the resources requested by the application to available system resources. If any of the requested objects cannot be honoured, the entire test fails. Once the system passes the admission test it goes into a real time mode (by calling a committing operation with a set of the required set of MPI/RT objects) in which the application uses system resources (e.g. network bandwidth and memory) in a controlled and predictable manner so that the communication operations complete within the QoS requirements and if a violation should occur, any user specified handlers defined for the events for these conditions will fire. Theses handlers may correct the system behaviour.

It is possible for the system to transit to another real-time mode within the same real-time phase by calling a committing operation with a different set of user-requested objects, so that only one real time mode is active at a time.

- **Finalization Phase.** The application enters this phase only from the Non-Real-Time Phase and after destroying all the created MPI/RT objects.

As seen above, the design of MPI/RT is based on a set of objects called Committable Objects that participate in the admission test performed in the committing function. Each of these objects has both object descriptors and attributes but does not have an object body, which should consist of implied system communication resources, until they are committed. Committable classes designed for those objects include the following (Kanevsky, Skjellum et al. 1997; 2002):

- **Containers;** a set of classes to create objects capable of storing references to selected MPI/RT objects and the operations for manipulating them.
- **Buffers;** the concept of buffers and their management emphasize that, in MPI/RT, all needed system resources are allocated outside of the application's real-time modes to reduce the dynamic memory allocation and the associated unpredictability during communication operations.
- **Buffer Iterators;** provides a mechanism for managing the order in which buffers are accessed. The standard specifies LIFO, FIFO, Sorted and Unsorted user selectable policies, in addition to other policies if the container is a vector container.
- **Instrumentation;** to measure the metrics that has to be measured automatically by the MPI/RT, e.g. the number of messages transmitted over a channel, MPI/RT provides probing objects that can be attached to the metrics in order to measure the change of the metric over time.

- **Channels;** channels are unidirectional, logical conduits through which data travels from one process to another with a particular QoS. An instance of the class of the channel represents a point-to-point channel end point and has two buffer iterators, for input and output that can be shared among several channels to form a collective channel in order to support collective operations such as broadcasting, besides the point-to-point channel.

For point-to-point and collective channels, MPI/RT supports three models:

1. **Two-Sided (Send-Receive) Communication.** The application issues data transfer for two sides of the data exchange. Under this configuration, all channel end points must call the *starting* operation for the message transfer to occur.
2. **One-Sided (Push/Pull) Communication**. Only one side of the application issues data transfer operations. Under this configuration, all but one channel end points must call the *activation* operation, while the remaining channel endpoint call the starting operation for the message transfer to occur.
3. **Zero-Sided Communication.** It is characterized by the absence of any data transfer operations, at the user thread level, by all sides participating of the data exchange. There is no rendezvous with processes at the communication end points. In (Neelamegam 2002), it is stated that:

   "It is the responsibility of the underlying middleware to schedule data transfer depending on the application's QoS, which translates to data transfer calls to/from pre-specified application memories at pre-specified times. Zero-sided communication necessitates early binding and hence eliminates the delay caused by handshaking, synchronization, and system calls".

   Under the Zero-Sided configuration, messages are automatically transferred either at fixed time intervals for the time driven paradigm, or by event occurrences for the event driven paradigm.

- **Event Triggers, Receptors and Handlers.** During application real-time mode, MPI/RT supports both:

1. **Implicit events:** MPI/RT generated events, such as QoS error or a buffer iterator overflow. The user can receive these events by creating receptors with MPI/RT identifying names of those events.
2. **Explicit events:** by calling a raising function of the event trigger object and deliver it to a user created receptor object.

The trigger object has a QoS parameter describing the frequency by which the event is raised, while the receptor object's QoS parameter describes the requirements for the event delivery from a trigger to the receptor once an event is raised.

The handler objects specify the application-defined functions that are executed in response to receiving event notification by receptors; they can be invoked synchronously or asynchronously with respect to the receptor to which they are attached. The asynchronous handlers are scheduled by the receptor according to the handlers' QoS outside the context of the receptor thread, prior to processing the synchronous handler. Also, MPI/RT offers controlled event synchronization by providing a wait operation on a synchronous event.

2.2 The Client Server Paradigm

The client-server paradigm is the most common paradigm for distributed applications, and many other paradigms are built upon it. In this paradigm, shown in Figure 2-3, asymmetric roles are assigned to two collaborating processes. One process, the server, acts as a service provider, which waits passively for the arrival of requests from clients. The other, the client, issues specific requests to the server and awaits its response. This paradigm is the principal paradigm for the Internet as many Internet services are client-server applications. These services are often known by the protocol that the application implements, such as HTTP, FTP, and DNS (Liu 2001).

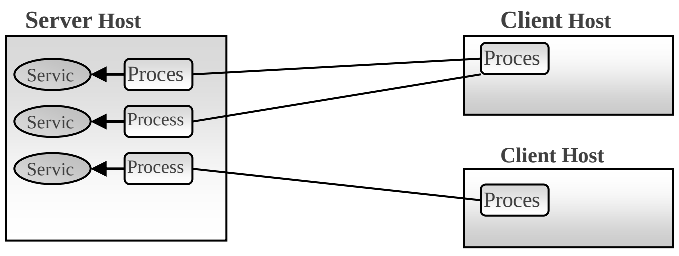


Figure 2-3 The Client Server Paradigm

Also, the middleware for database management systems; DBMS, and transaction processing, are built using the client-server paradigm and there has been much research to support real time and quality of service concepts within these middleware approaches; e.g. Beehive (Stankovic, Son et al. 1997). Although several commercial real time DBMS exists (e.g. Eaglespeed, TimeTen and PolyHedra), these systems have not yet solved all the problems of real time database systems (Son and Kang 2002). In such real-time middleware, many issues must be considered other than those considered in traditional ones; these issues include (Ramamritham, Son et al. 2004):

- **Data, Transaction and System Characteristics;** a transaction can be soft, firm or hard deadline transaction and can be periodic, e.g. periodic sensor measurements, or aperiodic one. Also, it can be static or dynamic. Consequently, the middleware must be able to satisfy the transaction's timing constraints, e.g. deadlines, earliest start times and latest start times, as well as data temporal consistency, i.e. how old a data item can be and still be considered valid.
- **Quality of Services and Quality of Data;** sometimes there is a need for making a trade off in the quality of data and accept a lower level of service in order to meet specified time constraints.
- **Scheduling and Transaction Processing;** the consideration of the tradeoff between the qualities of data vs. timeliness of processing is inherited in the scheduling of real time transactions. For hard deadline transactions, table-driven or rate monotonic priority assignment can be used but with many restrictions to keep the characteristics known a priori. In soft deadline transactions, the priorities are assigned according to the transaction time constraints rather than whether the transaction is CPU bound or I/O bound. Possible policies for scheduling of software transactions include:

  - Earliest deadline first.
  - Highest value first.
  - Highest value per unit computation time first.
  - Longest executed transaction first.

- **Distribution;** in many distributed applications, the real-time databases are not located on a single computer. Rather, they are distributed. This requires facilities for data replication, replication consistency, distributed transaction processing and distributed concurrency control.

2.3 The Peer-to-Peer Paradigm

In the peer-to-peer paradigm, the participating processes play equal roles, with equivalent capabilities and responsibilities. In this paradigm, shown in Figure 2-4, each participant, e.g. process A and process B, may issue a request to another participant and receive a response. This is different from the client server paradigm in that the client server paradigm makes no provision to allow a server process to initiate communication (Liu 2001).

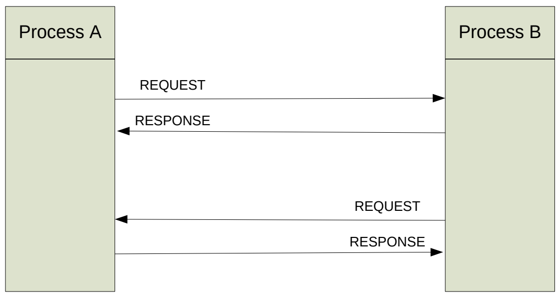


Figure 2-4 The Peer To Peer Paradigm

The main concept of the peer-to-peer computing is that each peer is acting as both client and server at the same time. Each peer is responsible for releasing and allocating the following (Loeser, Altenbernd et al. 2002):

- **The Processing Power;** the distributed computation is performed by a group of peers.
- **The Data Storage;** data is not owned by a particular member or server, but is passed around, flowing freely towards the end subscribers.
- **The Control;** each peer can offer the possibility of being controlled or illustrate monitored data.

Several peer-to-peer middleware technologies currently exist such as Gnutella, Jabber, FreeNet, and JXTA. Among those technologies, JXTA is the most significant because it was initiated to standardize a common set of protocols for building P2P applications. Also, Boeing adopted JXTA for the U.S. Army Future Combat System (Sun.com 2005). JXTA was introduced as an open source project by Sun Microsystems, the name JXTA was chosen as an abbreviation of the word "juxtapose", because to juxtapose is to put things next to each other, which is really what peer-to-peer is all about.

JXTA is a set of open protocols and implementations that allows any connected devices on the network ranging from cell phones and sensors to PCs and servers to communicate and collaborate in a direct P2P manner, where any peer can interact with other peers and resources directly, even when some of the peers and resources are behind firewalls. The JXTA reference implementation was in Java but other implementations are now available in other languages like C++ and C#. The main features of JXTA are (Sun Micro Systems Inc 2004):

- **Interoperability;** it can be used across different peer-to-peer systems and communities.

- **Platform Independence**; it can be used and implemented in multiple/diverse languages, systems, and networks.
- **Ubiquity;** it can be used on a range of digital devices such as PC's, PDAs, routers and servers.

The architecture of JXTA, shown on Figure 2-5, is divided into a three layered software stack as follows:

- **JXTA Core;** It is at the bottom of the stack, and it deals with the creation of peers and peer groups, ensures security within the network and manages the communication and routing between peers.
- **JXTA Services;** Provides services such as indexing, searching and file sharing.
- **JXTA Applications;** The actual applications that are built to make use of the JXTA P2P services such as messaging or document management systems.

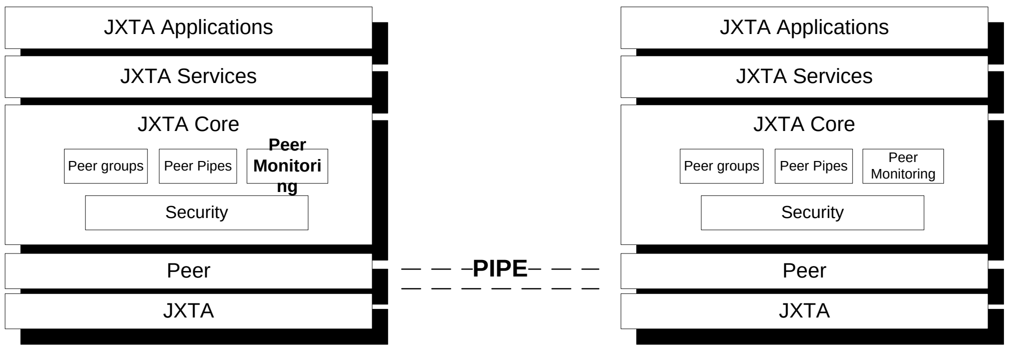


Figure 2-5 JXTA Architecture

Communication between peers in JXTA makes use of peers' advertisements. Once a peer connects to JXTA network, it publishes an advertisement that represent a summary of the peer (name, location, etc.) and what services it can offer. JXTA provides a discovery service that is used to locate an advert. Once obtained, a pipe connection can be established between the two peers. The pipe itself is a virtual communication channel that can be used for sending and receiving unidirectional messages asynchronously and does not belong to any one peer(Sun Micro Systems Inc 2004).

A broadcaster and listener form of communication among peers is supported in JXTA by using the Resolver. The Resolver is simply a query and response protocol, where the client sends an XML message targeted with a name and payload. The target peers look for the name of the message, process the payload, and return an answer if one is required. There is also another mode of operation for the Resolver called propagation; it is different from broadcasting, in that each computer may forward the message to others, rather than everyone listening to the same message all at once (Sun Microsystems Inc 2004).

The challenges of building real time applications on peer to peer networks and enhancing the current structure of JXTA to support such systems needs much research effort to support additional properties such as reliability, security and quality of service. This is due to the fact that peer-to-peer systems are (Chen, Repantis et al. 2005):

1. Inherently heterogeneous in terms of processor capacity, transmission loss rate, network inbound/outbound bandwidth, displays resolution as well as decoding software.
2. Peers are more dynamic than dedicated servers; they may fail or leave the system unexpectedly.

There have been some current effort to enhance the real time features of the JXTA peer-to-peer middleware; for example, in (Parker, Collins et al. 2004), the researchers analyzed the JXTA architecture for near real-time applications developments. They identified and tested the XML protocol handling and the Resolver service. The results from the Resolver service tests indicate that this component is suitable for near real-time applications, as it presented no unacceptable delays into the system performance. But on the other side, the results of the XML protocol handling tests indicated that it is not suitable for near-real time applications, as it introduces delays that are not within acceptable time limits.

2.4 The Message System Paradigm

This paradigm is an elaboration of the basic message-passing paradigm. In this paradigm, shown in Figure 2-6, a message system serves as an intermediary among separate, independent processes. The message system acts as a switch through which processes exchange messages asynchronously in a decoupled manner. A sender deposits a message with the message system, which forwards it to a message queue associated with each receiver. Once the message is sent, the sender is free to move on to other tasks. This paradigm can be classified into two subtypes as follows (Liu 2001):

I-The Point-to-Point Message Model

The point-to-point message model handles messages intended for a single receiver. Within a point-to-point message system, the messaging provider establishes queues to help ensure that a message is delivered to only one receiver only once.

II-The Publish-and-Subscribe Messaging Model

Publish-and-subscribe systems handle messages intended for multiple receivers. Publish-and-subscribe systems support the sending of messages to a destination by defining it as a topic or event. Applications interested in the occurrence of a specific event may subscribe to messages for that event. When that waited event occurs, the process publishes a message, announcing the event or topic. The message system is responsible for distributing the message to all the subscribers. This model offers a powerful abstraction for multicasting or group communication.

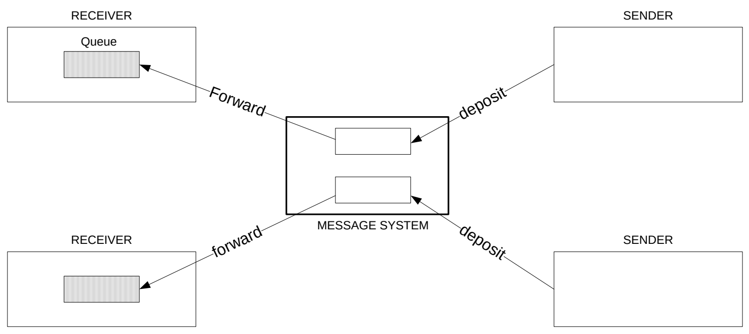


Figure 2-6 The Message System Paradigm

There are many examples on implementing this model as a middleware including Microsoft's Message Queue (MSQ) and Java's Message Service (JMS), but those systems were not targeted to the real-time domain.

One of the most important efforts to build a real-time message based middleware is the Data Distribution Services middleware, DDS. DDS is a formal standard middleware specification published by the OMG group (The object Management Group (OMG) 2007). DDS specification is not targeted to a certain platform or language, although it has implementations mainly in Java and C++, but other implementations in other languages are possible.

DDS targets mainly real-time systems; the API and QoS are chosen to balance predictable behaviour and implementation efficiency/performance. DDS is built on the idea of global data space of data objects that any entity can access, where the data object is uniquely identified by its keys and topics and the global data space itself is identified by its domain id; each publisher/subscriber must belong to the same domain to communicate. The architecture of DDS, shown in Figure 2-7, consists of the following entities:

- **Domain Participant;** this represents the publisher/subscriber holder object within a domain.

- **Publisher**; to publish the data, where the data could be of different types.
- **Data Writer;** used by the participant to communicate the value of and changes to data of a given type published by the publisher.
- **Data Reader;** used by the participant to access the typed received data and deliver it to the subscriber.
- **Subscriber;** receives the published data and makes it available to the participant.
- **Topic;** represents an association of a compatible Data Writer object, i.e. publication, with Data Reader objects, i.e. subscriptions. This association is by a name, a data type, and QoS related to the data itself.

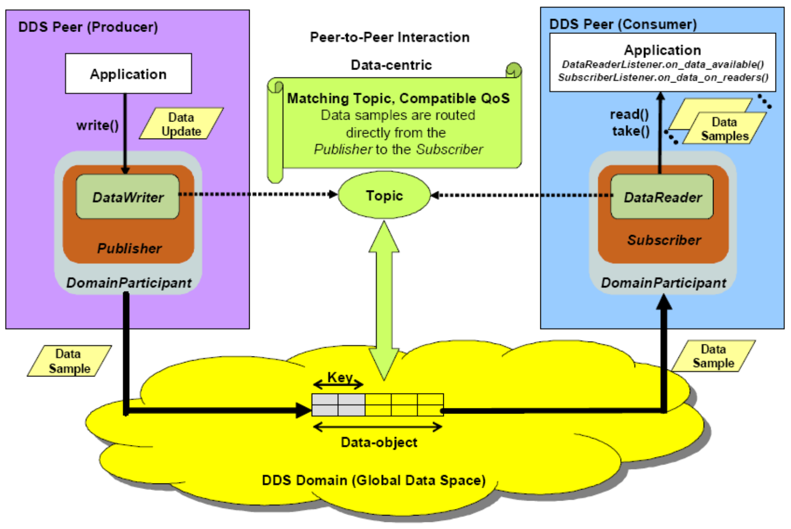


Figure 2-7 DDS provides a relational data model. The middleware keeps track of the data objects instances, which can be thought of as rows in a table(Pardo-Castellote, Innovations et al. 2005).

Each one of these entities can be configured through a corresponding specialized set of QoS policies, notified of events, and support conditions that can be waited upon the application.

Since the DDS discovery is spontaneous and decentralized, the topics can dynamically change over the lifetime without any administrative impact, where end-points are discovered automatically, and dynamic dataflow established in a plug-n-play fashion, which means DDS is suitable for scalability.

As DDS is targeted for real-time systems, its design supports a set of features, in a form of QoS policies, to enhance its performance. Some of these features include (Pardo-Castellote, Innovations et al. 2005; The object Management Group (OMG) 2007):

I-Predictable Delivery

In order to have predictable delivery of messages, the following QoS features are supported:

1. **DEADLINE:** It is defined as the maximum duration within which a Data Reader expects a data object instance to be updated.
2. **TIME_BASED_FILTER**: a minimum separation value that allows a Data Reader to ask for an un-sampled set of the values of the data object.
3. **TIME LATENCY_BUDGET**: A maximum acceptable delay from the time of writing a data by a publisher to the time of receiving it by the subscriber.

II-Delivery Ordering of Received Samples

In order to have a predictable delivery of messages, the following QoS features are supported;

1. **DESTINATION_ORDER**: To control the criteria of determining the logical order among changes made by different publishers to the same data object, either by a reception timestamp or by sending timestamp.
2. **PRESENTATION:** To specify how a coherent set of changes of a data object made by a single publisher are presented to a subscribing application.

III-Transport Priority

This includes the important QoS feature; **TRANSPORT_PRIORITY**, to allow the application to transport messages with different priorities**.**

IV-Resource Management

To manage the resources of the system in a predictable manner, the following QoS features are included:

1. **RESOURCE_LIMITS:** Specifies the resources that the middleware can utilize in order to meet the requested QoS.
2. **HISTORY**: Specifies how the middleware should behave when the value of a data changes before it is successfully communicated to one or more consumer.

V-**Status Notifications**

DDS support a number of status changes that can trigger a listener invocation on one of the DDS entities, e.g. OFFERED_DEADLINE_MISSED, SAMPLE_LOST OFFERED_INCOMPATIBLE_QOS, etc.

2.5 Remote Procedure Call Paradigm (RPC)

This paradigm was generated as a solution for the requirement of an abstraction that allows distributed software to be programmed in a manner similar to conventional applications running on a single processor. In this paradigm, inter-process communication proceeds as procedure calls which are familiar to application programmers. As shown in Figure 2-8, a process, A, wishes to make a request to another process, B, which may reside on another machine; this can be done by making procedure call by process, A, to the other process B. The remote procedure call involves (a) passing off a list of argument values to the procedure executing on the remote machine by process B, (b) at the completion of the procedure, process B returns a value to process A (Liu 2001).

The only well-known standard of this paradigm is the Open Group Distributed Computing Environment DCE's RPC (Opengroup). DCE's RPC is a language and platform independent middleware that runs on most major computing platforms and designed to support applications in heterogeneous hardware and software environments. It supports a limited set of services including thread management and distributed time services, but it has no support either for real time systems or embedded systems. The last version of this standard was released in 1997.

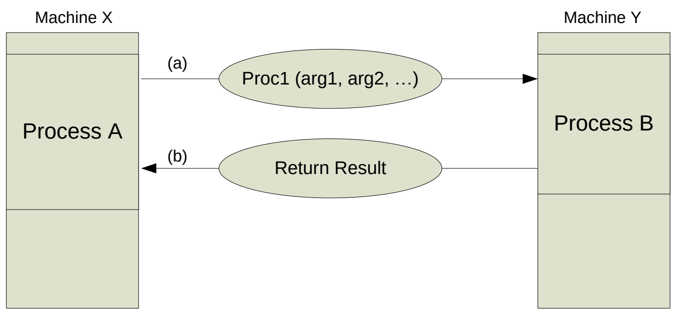


Figure 2-8 The Remote Procedure Call Paradigm

2.6 The Distributed Object Paradigms

In centralized object oriented software development, the software system can be built as a set of objects, where these objects communicate and coordinate among themselves on the same machine, in order to provide the required functionality of the system. The Distributed Object paradigm was introduced as a natural extension of object oriented software development, by enabling the communication among objects that reside on different machines in order to build a distributed system. In this paradigm the applications can access objects distributed over a network, where these objects provide methods that can be invoked to obtain access to services. This paradigm is considered currently the most active field of research in real-time middleware.

The distributed object paradigm is built upon the client-server paradigm and can be represented by several models. The main models of this paradigm are discussed briefly here. Then, in the next chapter, a detailed discussion of the most widely used two models (ORB and RMI) is presented.

2.6.1 Remote Method Invocation (RMI)

This type is the object-oriented equivalent of the remote method call. As shown in Figure 2-9, a process invokes the method in an object, which may reside in a remote host. As in RPC, arguments can be passed with the invocation. Java RMI is the most known example of this paradigm (Liu 2001).

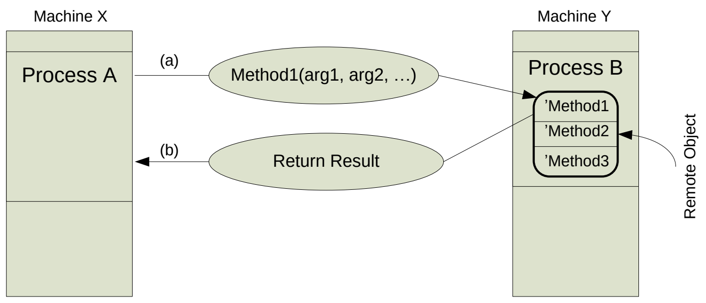


Figure 2-9 The Remote Method Invocation Paradigm

2.6.2 The Network Service Paradigm

In this paradigm, shown in Figure 2-10, service providers register themselves with directory servers on a network. A process desiring a particular service undertakes the following steps; (a) it contacts the directory server at run time; then (b) if the service is available on it, it will be provided a reference to the service; finally in (c) the requestor can access the service using this reference. In this manner, this paradigm is an extension to the remote method call paradigm. The difference is that the service objects are registered with a global directory service, allowing them to be looked up and accessed by service requestors on federated network. Java's Jini is a middleware example that is based on this paradigm , but, to our knowledge, it does not support real-time features.

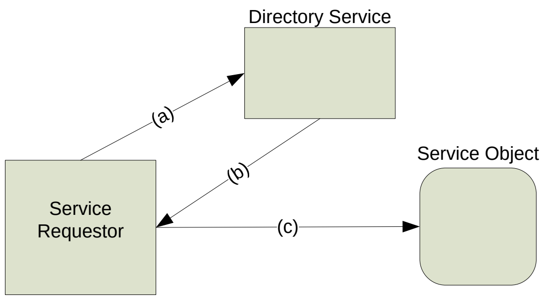


Figure 2-10 The Network Service Paradigm

2.6.3 The Object Request Broker Paradigm (ORB)

In this paradigm, see Figure 2-11, an application issues requests to an object request broker (ORB), which directs the request to an appropriate object that provides the desired service. Again, this paradigm is close to the remote method invocation paradigm, the difference is that the ORB acts as an intermediary which allows the object requestor to potentially access multiple remote (or local) objects, even when those objects were heterogeneous. This paradigm is the basis of the CORBA (Common Object Request Broker Architecture) Specifications, specified by the OMG (Object Management Group). Many implementations of the CORBA specification is available including Inspire's Visio Broker, Java's Interface Development Language, Orbix's IONA, and TAO from the Object Computing, Inc. (Liu 2001).

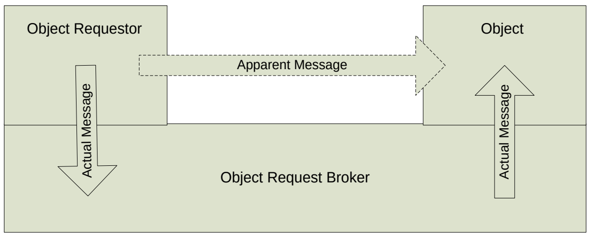


Figure 2-11 The Object Request Broker Paradigm

2.7 Component Oriented Architecture (COA) Paradigm

Component Oriented Architecture is based on set of well-known object oriented programming practices such as encapsulation, and separation of concerns. Applications built using the COA paradigm, are decomposed into *Components* that have clearly defined responsibilities and interact with one another through standardized interfaces, Figure 2-12. The components are hosted in a *Container* that is responsible for managing component lifecycle, and facilitates communication between components, either by providing a *Service Locator* or by *Dependency Injection* design patterns. Also, the Container architecture provides a runtime engine with a set of common services that all components use. These services include common interfaces to functions such as security, transactions and database connectivity. Examples of this paradigm include Microsoft's COM, EJB (Plášil and Stal 1998).

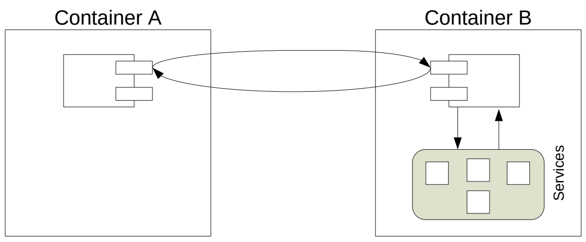


Figure 2-12 Component Based Architecture Paradigm

2.8 The Application Server Paradigm

The application server model is known as the second-generation client-server architectures or three-tier architecture model (RENAISSANCE Consortium 1997). In this paradigm, shown in Figure 2-13, application logic is not tightly connected with either of the data or objects management or presentation, and is implemented in a separate layer; i.e. the application server is a middle tier layer. The Application Server is the middleware responsible for providing the access to objects or components to the clients requesting it.

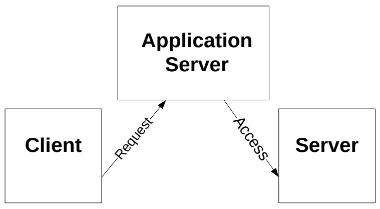


Figure 2-13 The Application Server Paradigm

Web Services adopts this paradigm to provide the next wave of web based computing. Web services were defined by World-Wide Web Consortium (W3C 2010) as follows :

> *"A Web service is a software system identified by a URL, whose public interfaces and bindings are defined and described using XML. Its definition can be discovered by other software systems. These systems may then interact with the web service in a manner prescribed by its definition, using XML based messages conveyed by Internet protocols. "*

The core web services standards are the following (Jammes and Smit 2005):

- **Web Service Description Language (WSDL);** for abstract description of the service interface.
- **XML Schema;** to define the data formats used for constructing the messages addressed to and received from services.
- **SOAP;** protocol for transferring the service related messages.
- **WS-Addressing;** concentrates message addressing information into the header of the SOAP message envelope, to enable message contents to be carried over any transport protocol.
- **WS-Policy;** to express policies associated with the service.

- **WS-Metadata Exchange;** to enable dynamic retrieve of metadata associated to a web service.
- **WS-Security;** optional set of mechanisms to ensure end-to-end message integrity, confidentiality and authentication.

One of the great strengths of Web services is that they are atomic transaction-based. However, the transaction-based has its limitation as each transaction incurs the overhead of the request response protocol and requires a long time in processing and causes much network traffic. Also, there is no mechanism to subscribe for a "push" service; a client must pull the values it needs. Moreover, the transforming of messages into XML format and XML parsing processes is a time consuming process. This requires extending the Web services to overcome these limitations to use it in real time applications (Morse, Brunton et al. 2004).

2.9 The Tuple Space (Object Space) Paradigm

This paradigm, shown in Figure 2-14, assumes the existence of logical entities known as object spaces where the participants of an application converge in a common object space. A provider places objects as entries into an object space, and requesters, who subscribe to the space, access these entries. The object space provides a virtual space or meeting room among providers and requestors of network resources or objects in a manner that hides the details involved in resource or object lookup needed in other paradigms (Liu 2001).

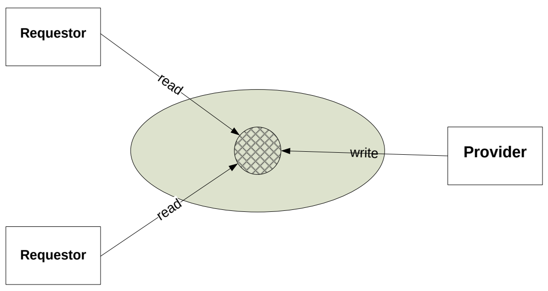


Figure 2-14 The Object Space Paradigm

Several models can be used to implement Tuple-Spaces in distributed memory systems including (Wells, Chalmers et al. 2000):

- **Centralized Systems;** all the tuples are stored on a single processing node.
- **Hashing System;** contents of tuples are used to allocate them to particular processors
- **Partitioned Systems;** tuples with a common structure are allocated to a specific processor.

- **Fully distributed;** any tuple may reside on any possible node.

The Tuple space paradigm was first proposed as a part of the Linda coordination language for parallel and distributed processing, where the data is represented by elementary data structures called tuples, in a form of shared object, and the shared memory is a multiset of tuples called tuple space. Each tuple is a sequence of typed fields such as <"data", 22, 4.5>. Processes in the system have a handle to the tuple space and they can do a basic set of pre-defined operations in the system including:

- *out(t)* operation to add tuple *t* to the tuple space.
- *in(p)* operation to remove a set of tuples by, where *p* is a pattern.
- *rd(p)* operation to read a set of tuples by, where *p* is a pattern.

Where both *in(p)* and *rd(p)* are blocking operations.

Another extension of this model is provided by using a pair of asynchronous operations *inp(p)* and *rdp(p)*, which return null if no matching tuple exists in the tuple space.

Although it is not quite as efficient as message passing systems, Tuple space provide a simple, yet powerful mechanism for inter-process communication and synchronization which makes it easier to write and maintain for the following reasons (P. Wyckoff, S. McLaughry et al. 1998):

- **Destination Uncoupling;** the creator of the tuple requires no knowledge of the future use of the tuple, nor its destination, so Tuple space communication is fully anonymous.
- **Space Uncoupling;** the addressing scheme used for addressing tuples in Tuple space paradigm is the associative addressing rather than a physical one. So, it provides a globally shared data space for all processes, regardless of the underlying machine or platform.
- **Time Uncoupling;** each tuple has a life span independent of both the process that generated it, and the process that may read it. This independence enables processes to communicate seamlessly even if they are not available at the same time.

Tuple-Spaces have found their way to the commercial implementations; two such implementations are compared in (Wells, Chalmers et al. 2000); Java Spaces from Sun, and TSpaces from IBM. Both are using a centralized implementation of the Tuple space. Each of them provides a Java object oriented implementation of the original

Linda model (using different names of the original instructions) with some extensions of their own. Some common extensions in both products include:

1- Supporting the blocking operations with a time out.
2- Supporting the execution of a set of Tuplespace-operations as a transaction.
3- Giving an expiry date for the tuples (leases in JavaSpaces).

In JavaSpaces, tuples are created from classes that implement the marker interface `Entry`, where only the public fields are considered, and the tuples are transmitted by serialization of the public fields only (Sun Microsystems Inc 2003).

On the other hand, tuples in TSpaces consist of a number of Field objects classes, e.g. `String.Class` that can be transferred using the standard Java serialization. Also, it extends the original Linda's operations by a rich set of operations for deleting a tuple, input/output operations of multiple tuples and operations to specify tuples by means of tuple-ID rather than the usual associate addressing mechanism. Moreover, the TSpaces extended the querying of a tuple space to 4 basic methods: Matching, Indexing, And-ing, and Or-ing. Also, TSpaces supports the concept of handling events at the source process, when a tuple is read or taken by a target process.

In general TSpaces is more simpler to run and needs only a single server process to run on the network, while JavaSpaces is itself a basic service of a larger middleware for heterogeneous system called Jini, and its network support is provided by the Java RMI. This means that it cannot work without starting all those services.

From a real-time point of view, in 2004, Sun designed, a new soft real-time system to manage telemetry data for *Formula-1* racing cars in real time, using software implemented using Java and Jini technology with the aim of achieving the required performance (Sun Microsystems Inc 2004), to support multiple platforms, such as Linux and Windows, and to provide high availability and location transparency of components.

A common criticism of the Linda model is that it is inefficient and subject to unpredictable performance. The simplicity of the model hides the underlying complexity of the required data sharing and communication. In order to overcome some of these problems the authors of (Wells, Chalmers et al. 2000) proposed an extended version of the original Linda model, the new model adopts a fully distributed Tuplespace model, called eLinda, which is implemented in Java.

2.10 The Collaborative Application (Groupware) Paradigm

In systems such as Virtual Organization and Video Conferencing, multiple parties collaborate in order to provide a particular service. These systems are using the Collaboration paradigm, Figure 2-15, to implement the required sharing of a common state. According to the placing of the common state of the system, the collaboration can be implemented in two ways (Liu 2001):

- **The Message Based Groupware;** by using messages to propagate the state to each local copy of the shared state.
- **The Whiteboard Groupware;** the shared state is kept in a central location where collaboration parties can access it.

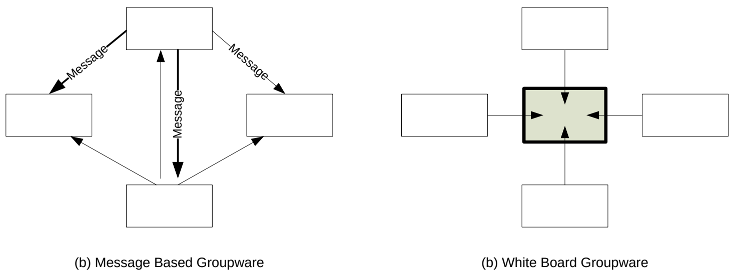


Figure 2-15 The Collaborative Paradigm

One of the most interesting soft real time middleware approaches using this paradigm is H.323, which is a protocol suite published by International Telecommunications Union (ITU) for the first time in 1996 and the last version (Ver. 5) was published in 2003. The H.323 supports media (voice and video) communication and data collaboration over networks that do not provide a guaranteed quality of service. The H.323 itself is an umbrella specification because it includes various other ITU standards such as Real-Time Protocol (RTP) and Real-Time Control Protocol (RTCP), with additional protocols for call signaling, and data and audio-visual communications.

The architecture of the H.323 includes mainly a set of collaborating components (Mark A. Miller 2005) as follows:

- **Terminals;** represents the end device of every connection, it provides real-time two-way communications with another H.323 terminal, GW or MSU.
- **Multipoint Control Unit (MCU);** allows three or more H.323 terminals to connect and participate in a multipoint conference.
- **Gateway Device (GW);** to establish the connection between the H.323 terminals with terminals which belong to networks with different protocols.

- **Gatekeeper Device;** to translate between telephone number and IP addresses. Also, it manages the bandwidth and provides mechanisms for authentication and registration.

Communication Door is another soft real-time middleware communication middleware that uses the collaboration paradigm. It enables developers to build custom multimedia streaming solutions via broadband Internet. It supports three major features:

a. Real-time synchronization of contents among web browsers connected to the same web application.
b. Real-time voice and video communication.
c. An HTTP-based push server that can autonomously send differences in data maintained by the server to web browsers connected to the application.

The architecture of Common Door consists of the following collaborating parties (TABUCHI M, NAKAJIMA K et al. 2004):

- **ASP Applications;** web pages as interface for the user to access the application.
- **ActiveX Controls;** hosted in web pages to send and receive data and media with the Push server.
- **Web Browser;** to render the web pages.
- **Push Server;** sends the updated media/data to ActiveX controls on the web browsers via HTTP in real time.
- **Audio/Video Relay Server;** to relay audio/video streams sent by an ActiveX control one browser to other ActiveX control on another browser.

Another example of this paradigm is found in (Gong, Kulikowski et al. 1997), where a real-time collaborative system was presented for medical image analysis and distributed radiological reporting. The system was based on AI methods for intelligent control and it was implemented using Java and CORBA standards.

2.11 Summary

In this chapter, proposed structural paradigms of the middleware technologies are presented from the least abstracted to the most abstracted. The basic elements of each paradigm structure and its advantages for building distributed systems are presented, and examples of existing real-time implementation of some of those paradigms are discussed. We have seen that although that the Java language has found its way for implementing many of the middleware technologies; there are no commercial Java implementations of any of these real-time technologies. Hence, there is a need to study the reasons that inhibit the use of the Java language in building real-time systems, particularly the middleware solutions, and we need to survey the recent research made to use the Java language for building real-time middleware solutions. Hence, in the next chapter, we present the limitations of the Java language; then, we cover how the Java language adopted the RMI as the distributed object paradigm for building middleware solutions, and how the real-time Java community addressed the RMI middleware technology to build real-time middleware solutions using the real-time specification for Java (RTSJ). In addition to RMI, we will cover the work done on CORBA, as a language independent implantation that supported in Java and we will show the efforts done to implement it using RTSJ.

Chapter 3

Real-time Middleware in Java

In the previous chapter we presented general overview of the different middleware paradigms, and the discussed some real-time implementations of these paradigms. In this chapter, we aim to focus on the recent research efforts on building real-time middleware using the Java language.

Java supports mainly two middleware-frameworks to program distributed system: Remote Method Invocation (RMI) and CORBA. Both frameworks implement the Distributed Object application paradigm, where a typical client application gets a reference to one or more remote objects in the server and then run methods on them. Both frameworks are used as a base for building many other distribution and middleware frameworks in Java (e.g. Jini and EJB). Both middleware, just as the Java language itself, were originally implemented without any consideration to support real-time distributed systems.

In this chapter, both frameworks are presented with concentration on the current efforts to support real-time features. So, we first present the basic features of the Java language that helped it to be one of the most used programming languages at the moment. Then we present the factors that made it less appropriate for building real-time systems and we will discuss the new models and features that have been presented in the RTSJ to overcome these limitations. After that we will discuss in detail the Java RMI structure and the properties that make it not applicable in distributed real-time systems. Then we will review the research that has been done to integrate the RTSJ with the RMI and distributed systems in general.

3.1 Java and Real-Time systems

Java is a modern object oriented programming language that has many features which make Java very efficient in building software systems in general, and distributed systems in particular. The success of Java in building distributed systems and middleware solutions is due to its packages that simplify the network programming and abstracts a lot of its complexity.

Java has many enhanced features and properties that enhance the speed and quality of the software development, these features include (Wellings 2004):

- It has a high level of abstraction, and it is *easier/faster* to master than other languages (e.g. C++ or Ada); this makes it have a faster learning curve that allows for increased programmer productivity.
- As it using object oriented principals of accessing objects, it is relatively secure, keeping software components protected from each other.
- It includes high levels of dynamism, by allowing dynamic loading of classes and supporting dynamic creation of objects and threads.
- It supports component integration and reuse as it supports building reusable software packages and libraries.
- It is platform independent and supports application portability which makes it very efficient for distributed applications.
- It provides well-defined execution semantics and supports strong typing. Hence, it offloads many tedious and error-prone programming details.
- It has a powerful and portable standard library.
- Its byte code representation is more compact than native code.
- It has portable support for concurrency and synchronization.

As Java has met a lot of success in building commercial software systems, there was a strong motivation in the real time community to consider using Java in real-time computing. However, it has not met the same success in building real-time systems. So, there was a requirement to analyze the language to discover the required real-time features that are missing of the Java language as well as its current features that makes it non-usable in many real-time systems.

A set of the limitations that made Java unsuitable for use in real time systems includes (Sun Microsystems Inc 2005):

- Its memory management models offer unpredictable latencies due to using garbage collectors.
- The inadequate scheduling control.
- Unpredictable synchronization delays.
- Very limited timer support.
- No support for asynchronous event handling (AEH).
- No support for safe asynchronous transfer of control (ATC).

In the last few years, many efforts have been made by researchers in real-time systems to overcome these limitations. These efforts led to the RTSJ; **R**eal **T**ime **S**pecification for **J**ava (first published in *January 2002*); that made using Java in real time systems a reality (*first commercial implementation in 2003).* RTSJ offered many features that overcome the limitations of original Java including (Wellings 2004; Sun Microsystems Inc 2005):

- **New Memory Management Models;** RTSJ offers the use of memory regions (immortal memory and scoped memory) that are not subject to garbage collection. Where immortal memory region holds objects without destroying them until the program ends, and scoped memory region is used only when the process works within a particular scope of the program (e.g. a method), where the objects are destroyed automatically when the process leaves this scope.
- **Access to Raw Physical Memory**; RTSJ offers completely Java securely protected direct access to physical memory instead of linking to native code libraries. This means that device drivers can be written completely in Java.
- **Strong Guarantee on Real Time Thread Semantics;** RTSJ introduces two new types of threads: real-time threads, and no-heap real-time threads (a thread that cannot be interrupted by garbage collector). These threads offer more precise scheduling than standard Java threads. They have 28 levels of priority and unlike Java their priority is enforced.
- **Higher Resolution Time Granularity and Clock**; RTSJ offers several ways to specify relative and/or absolute high-resolution (Nano-second accuracy) time.
- **Support for Asynchronous Event Handling (AEH);** RTSJ allows developers to schedule the response to asynchronous events in order to avoid disrupting the temporal integrity of the rest of the real-time application.
- **Support for Asynchronous Transfer of Control (ATC)**; RTSJ provides a carefully controlled way for one thread to safely interrupt another thread.
- **Support for Schedulable Objects and Scheduling**; RTSJ has two types of schedulable objects: real-time threads, and asynchronous event handlers. Where each schedulable object can be associated with attributes such as release parameters, scheduling parameters, memory parameters, and/or processing group parameters. RTSJ also defines the priority scheduler that uses the fixed priority policy where the processing resource is always given to the highest priority runnable schedulable object allocated to the processor.
- **Support for Synchronization and Resource Sharing;** to bound the blocking suffered by schedulable objects, RTSJ supports two priority inheritance algorithms:

simple *priority inheritance* and *priority ceiling emulation inheritance*. Also, RTSJ provides a non-blocking communication mechanism to avoid the unpredictable interactions with the garbage collection (e.g. in case of schedulable objects that communicate with non-real-time objects).

Distributed real-time systems were not addressed in the RTSJ and it was deferred for later stages but, after the success of RTSJ, many researches targeted the Distributed Real Time Specification for Java (DRTSJ) (Clark, Jensen et al. 2002).

3.2 Overview of Java RMI System Architecture

RMI is a Java mechanism to call methods of objects that do not exist in the same virtual machine in the same way as if they were local methods. The architecture of RMI aims to hide most of the network communication implementation details and to allow programmers to build safe and robust distributed Java programs with *nearly the same* syntax and semantics used for non-distributed programs.

In RMI, see Figure 3-1, there is always a server, which acts as a service provider, and a client that act as a service receiver. Both server and client are Java objects that can interact asymmetrically. The server must document the description for the remote service it provides as an interface that extends the Java Remote interface. From this description, the rmic compiler creates additional classes, which, are used, by both the client and the server as proxies to hide the communication details[1]. To be accessible by clients, the server should register the service's object in the RMI registry, which acts as a naming server to hold references to the remote services. Once the client needs to use the remote object, it looks up the registry to get a remote reference to this object and then uses this remote reference to invoke the required method of the remote object (Boger 2001).

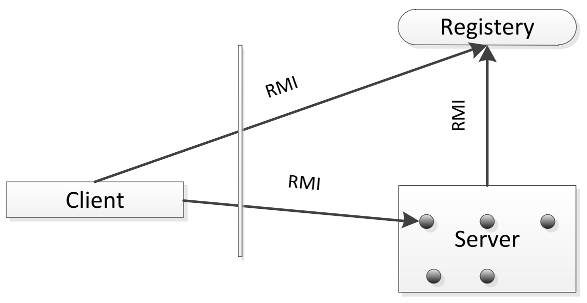


Figure 3-1 The RMI Communication mechanism

[1] The wire protocol starting from Java 2 SDK uses reflection to make connection to the remote service object instead of using the skeleton helper classes (server side proxy) generated by the rmic in JDK 1.1 and JDK 1.1.

The RMI structure can be divided mainly into three sets of models as follows (Borg and Wellings 2003):

- **The Programming Model;** this includes the interfaces and exceptions required to define the remote objects and its failure semantics.
- **The Implementation Model;** the transport mechanisms by which one Java platform can request and access objects of another Java platform.
- **The Development Tools;** one example of such tools is *rmic*, which takes the server, objects and generate the proxies required to facilitate the communication.

In the following sections the details of the structure is presented followed by a discussion of its problems and requirements to act as real time middleware.

3.2.1 RMI Layers

The RMI architecture is built basically from three architectural layers, shown in Figure 3-2. The architecture of each one of these layers is presented here.

A- The Stub Layer

The stub at the client side acts as a proxy to forward the method invocations to the server's JVM that holds the object. It is responsible for:

1- calling the remote reference layer,
2- informing the remote reference layer by starting/ending of invocation,
3- marshalling passed arguments and un-marshalling the return value, or exception from a marshal stream.

A corresponding layer, skeleton layer, used to be responsible to communicate with the stub to carry on the invocations at the server side. From Java 2 implementation of RMI, the class responsible of this layer became obsolete and RMI now uses reflection to do its functionality dynamically.

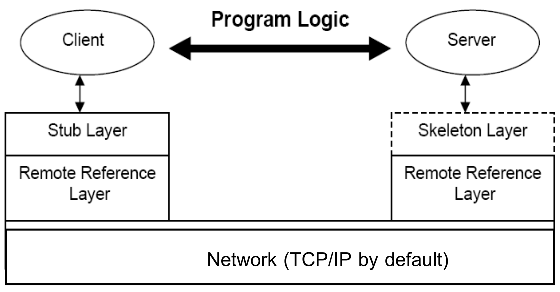


Figure 3-2 RMI Layers

B- The Remote Reference Layer

This layer is responsible for the interpreting and managing references made from clients to server's objects across JVM's through the `RemoteRef` object, that represent the link between the client and the remote object. The Stub objects use the `invoke()` method in `RemoteRef` to forward the method call. Each remote object implementation chooses its own remote reference subclass that operates on its behalf (Sun Microsystems Inc 2004). In JDK 1.1 implementation of RMI, only unicast point-to-point connection was implemented using `UniCastRemoteObject` class for creating and exporting remote objects, where the remote object should be instantiated and exported first to the RMI system before a client can use it. Later, in Java 2 SDK implementation of RMI, a support for *activatable* remote object is added to enable the RMI system to instantiate the object and restore its state from a disk file into the memory before accessing it (Sun Microsystems Inc 2003; Sun Microsystems Inc 2004).

C- The Transport Layer

The transport for Java RMI can be divided into four basic abstractions as follows (Sun Microsystems Inc 2004):

- **An End Point;** it is the abstraction used to denote an address space or JVM.
- **A Channel**; represents a virtual connection between the local and the remote end points.
- **A Connection;** is the abstraction for performing input/output data.
- **The Transport;** it is an abstraction used for managing the channels.

The responsibility of the transport layer include building, managing, monitoring and maintaining the stream based network connection between the client and server endpoints (Sun Microsystems Inc 2004).

The current implementation of Java RMI is using TCP sockets as a default implementation of the transport layer. On top of TCP/IP, RMI supports two protocols:

1- JRMP; Sun's wire-level Remote Method Invocation Protocol between RMI objects.

2- RMI-IIOP; Internet Inter-Orb protocol that provides interoperability of RMI objects with CORBA Objects. It is available starting from the Java 2 platform standard edition, version 1.3.

Other types of connection semantics are possible at this layer other than the Unicast point–to–point (Sun Microsystems Inc 2004), such as the invocation to replicated object groups by using multicasting where a single proxy could send a method request to many replicates or caches simultaneously and accept the first reply. In (Krishnaswamy, Walther et al. 1998), the remote reference layer was extended to cache the remote objects at the client nodes, see Figure 3-3. In this approach, the client (C) at node (P_i) is allowed to transparently invoke remote objects (O) from node (P_s) independent of whether they are being cached. When a cached copy (O') of an invoked object is available, the invocation is executed locally; otherwise, the invocation is done remotely. In order to maintain the consistency of the replicated (cached) object copies in this approach, consistency protocols benefit from using flexible modified transport layer that supports multicasting communication semantic.

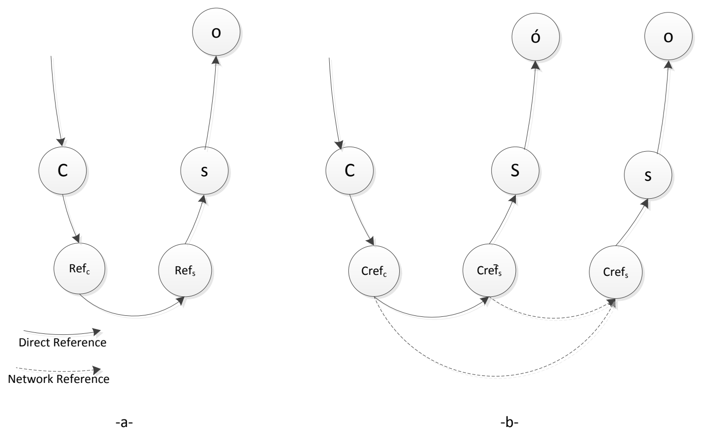


Figure 3-3. RMI Model. -a- without Caching -b- with Caching

3.2.2 RMI Distributed Garbage Collection (DGC)

Distributed Garbage Collector (DGC) is used to keep track of all the references used by an RMI-based application in order to free them when they are no longer used. The reference-counting garbage collection algorithm is used by keeping track of the live references within each JVM (Sun Microsystems Inc 2004). The DGC uses the leasing mechanism whereby remote references are leased for a period of time by the client holding the reference to overcome the problems appearing when a client crashes, or the network goes down. If the client does not renew the lease before it expires, i.e. using another call to the `dirty()` method, the DGC assumes that client no longer references the remote object. On the server side, the RMI runtime keeps track of the active clients in a Client Reference List. When a remote object is not referenced by any client it is removed from the RMI DGC, thus makes it possible to be reclaimed by the local garbage collector. The interface DGC abstracts the distributed garbage collector of objects at server side. It has two main methods: `dirty()` and `clean()`. The `dirty()` is invoked by the client's stub once instantiated, to lease the remote object at the server for a certain period, whereas the `clean()` is invoked by the stub when no more references to the remote reference exists in the client to let the server know that the client finished using the remote object (Sun Microsystems Inc 2004).

3.3 RMI Framework Implementation

Classes and interfaces used to implement the layers of RMI framework at both the client and the server are presented here with a brief description of the rules of each (de Miguel 2001; Sun Microsystems Inc 2004).

3.3.1 RMI Client Implementation

The main classes and interfaces to implement the client side of the RMI framework is shown in Figure 3-4 taken from (de Miguel 2001). The classes and interfaces according to the layers of the RMI are as follows (Sun Microsystems Inc 2004):

- **Transport Layer;** the RMI transport implementation includes the class `java.rmi.server.RMIClientSocketFactory`, which is the default resource provider for client socket (through its `createSocket()` method) that is used to send and receive RMI calls. The interfaces representing the Endpoint, Channel and Connection abstracts of the transport layer are implemented using the TCP protocol in the `TCPEndPoint`, `TCPChannel`, and `TCPConnection` classes.
- **Remote Reference Layer;** the main interface in this layer is the `RemoteRef` interface, it represents the abstract of this layer and it is realized in the `UniCastRef` Class where the execution of the two methods `newCall()` and invoke() create new or (reuse) connection and socket instances via `LiveRef` and the transport layer. When the method call ends, both the connection and the socket are destroyed after 30 seconds unless reused by another call.
- **Stub Layer;** the rmic compiler generates the Class-Stub as a subclass of the `java.rmi.server.RemoteStub`, which extends the Remote Object class that implements the
- **RemoteRef interface**; the RemoteRef represents the handle that contains the concrete representation of a reference and is used by the stub to carry out remote calls on the objects for which it is a reference.

The generated Class-stub acts as a surrogate that supports exactly the same set of remote interfaces defined by the actual implementation of the remote object. The Stub carries out a call to a remote object by implementing the `RemoteCall` interface, which includes basically the methods:

- `getOutputStream()`: returns the stream in which the stub marshals arguments.
- `releaseOutputStream()`: releases the output stream.
- `getInputStream()`: returns the stream in which the stub unmarshals results after ending the invocation.
- `executeCall()`: to execute the call.

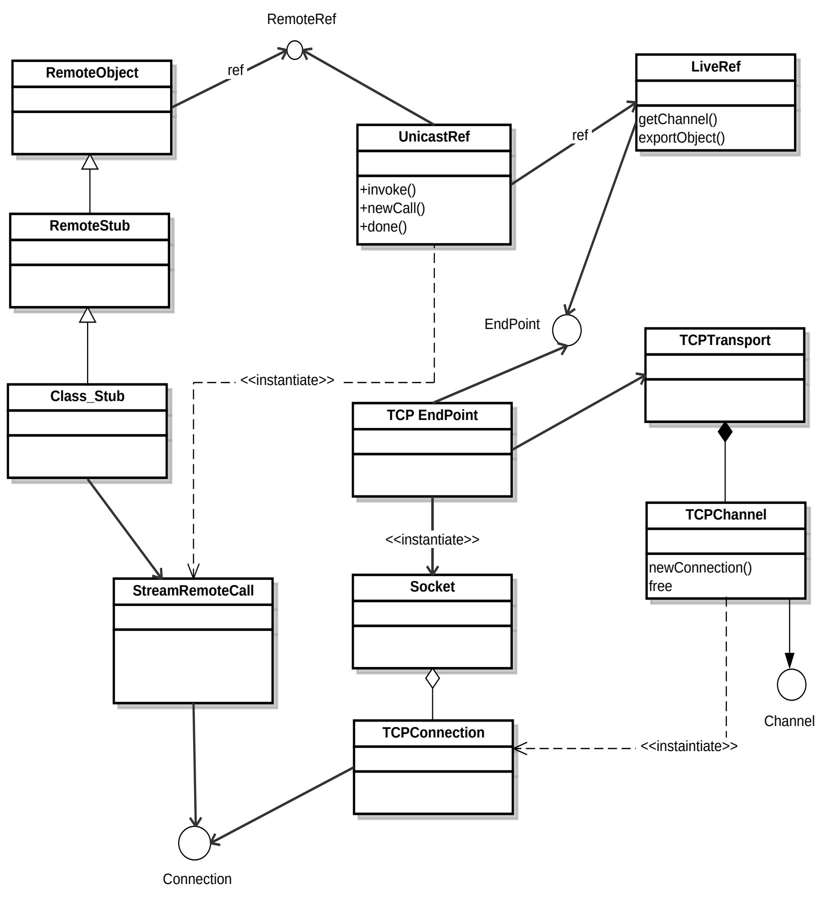


Figure 3-4

Classes of the Client in the RMI

3.3.2 RMI Server Implementation

The implementation of the server side is rather more complicated than that of the client side. This is due to the nature of the server that is responsible for both exporting the object for remote invocation and the execution and returning the result to the client. Figure 3-5 shows the most important classes and interfaces used in the implementation of the server side of RMI. The implementation elements according to the layers of the server layers are as follows:

- Transport Layer; the RMI transport and its implementation at the server side are identical to that used at the client side, except for a few differences that support some added server functionality. One such difference is that the `TCPTransport` class at the server side is an active class waiting for new connections requested by clients. Once a connection requested, `TCPTransport` spawns a new non-system thread (i.e. Java thread) and creates a new instance of the `ConnectionHandler` active class that is responsible of receiving the client calls and delegating them to the `UnicasetServerRef` at the reference layer. The `ConnectionHandler` delegates the invocation, but does not wait for the result.

- Remote Reference Layer; the hierarchy of this layer is close to its correspondent in the client side. However, in this layer, the `UnicastRef` class is extended into the `UnicastServertRef` Class that contains the static `exportObject()` method that is responsible for invoking the exportation function of the `TCPTrsnsport` class to do the exportation of the object. Also, `UnicastserverRef` has the `dispatch()` method that receives the client calls delegated by the `ConnectionHandler` class in the transport layer, and then it calls the skeleton to dispatch the invocation.

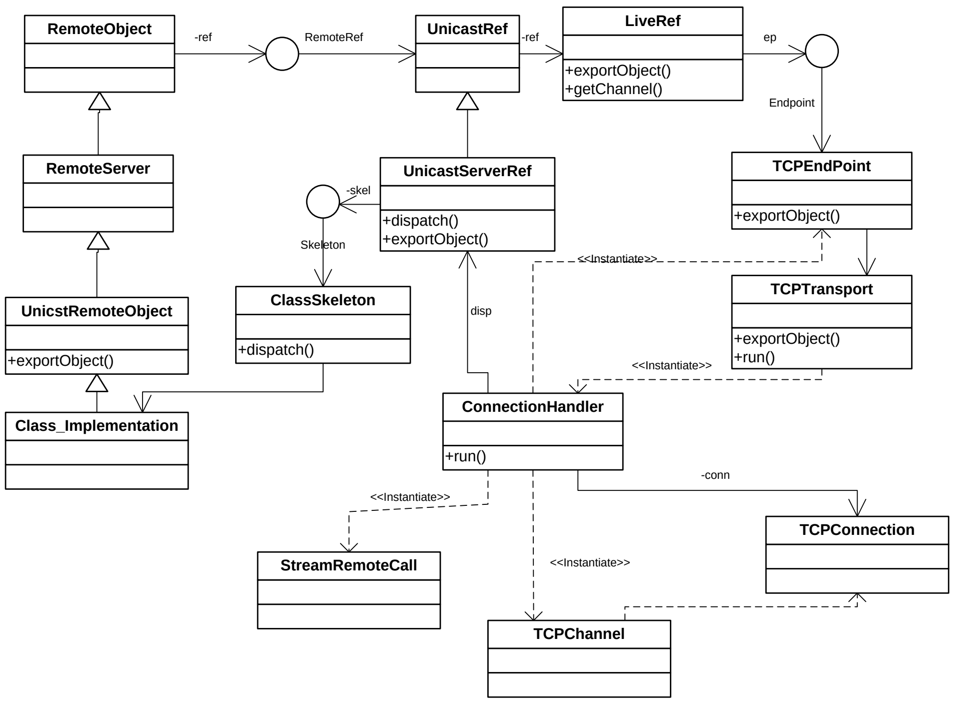


Figure 3-5 Static Diagram of the RMI's Server classes

- The Skeleton Layer; as it generates the stub for the client, the rmic compiler generates the Class-Skeleton that is used by `Class_Implementation` which extends the `UnicastRemoteObject` and is acting as the application implementation of the RMI server. The `UnicastRemoteObjec` at the server side differs slightly than the client side, as at the server side it extends the `RemoteServer` that defines the functions needed to create and export remote object. Once invoked method is executed at the server side, the skeleton returns the results using an instance of `StreamRemoteCall` that had been created during the invocation at the server side to implement the `RemoteCall` interface.

3.4 Problems and Requirements of RMI in RT Systems

To understand the problems inherited in the structure of the RMI middleware that makes it unsuitable to be used as a real time middleware, a classification of the problems and the requirements of it are presented here with respect to:

1. Java and JVM limitations.
2. RTSJ support
3. RMI tools
4. RMI programming model.
5. RMI implementation model.

Each of the above components suffers from some problems that make it unpredictable and unsuitable for distributed real time applications. Here the main problems in RMI are identified and possible solutions as offered by researchers are presented to solve these problems.

3.4.1 Java and JVM Requirements

Using the Java language in real time systems in general and distributed real time systems particularly, faces many challenges to fulfill and overcome some of the limitations in the language and the JVM specifications. Some of the requirements in regard to RMI are mentioned here.

A- Global Clock

RMI assumes the clocks at both client and server sides progress approximately at the same rate. This assumption is not strong enough and can be easily violated resulting in unpredictable behaviour. For example, the distributed garbage collection is dependent on the leasing mechanism in which the leasing period is defined by the client to reserve the remote object at the server, for a certain period of time. If the clocks at both sides are not progressing at the same rate, the remote object may be unpredictably removed resulting in a remote exception being raised when access is attempted (Wellings, Clark et al. 2001). To solve this problem JVM at both client and server should progressively synchronize their clocks.

B- Dynamic Java

As in centralized systems, it is currently unclear how the dynamic class loading fits into a real-time framework of RMI. Also, the dynamism resulting from polymorphism makes the real-time analysis hard or pessimistic (Borg and Wellings 2003).

C- Circular Referencing

The problem of absence of any facility in RMI to avoid the circular referencing in distributed systems makes it possible for the deadlock to occur. The following scenario for two objects A and B on different virtual machines describes this problem (Clark, Jensen et al. 2002);

> *"A client thread calls a synchronized method in remote object A, which calls a synchronized method in object B. The called method in object B then directly (or indirectly) calls a synchronized method in Object A."*

As JVM and RMI do not collectively support the notion of a distributed thread, the proxy thread executing the method on object A is not considered the same as the original client thread.

This same scenario, leads to another problem in the thread local-data; that the data read by the second call to object A will not be the data that is read by the original call. The solution to such a problem needs coordination among the JVM to globalize the identifiers of the references.

D- The Distributed Garbage Collector

The unpredictability of the garbage collection execution leads to latencies and priority inversion that must be avoided in real-time systems. To avoid the effects of the garbage collector there are two main solutions (Tejera, Tolosa et al. 2005):

- Use real time garbage collector (RT-GC)
- Use memory regions concepts to allocate memory out of the heap.

RT-GC cleans up memory in a predictable way, but at the cost of introducing overheads in program execution. Where, the memory region solution requires the programmer be aware of the lifetime of objects to group them in the same memory region, which is not easily done (Basanta-Val, García-Valls et al. 2004).

RTSJ supports the second solution by providing special thread types (e.g. RealTimeThread) and a memory model (e.g. immortal memory, scoped memory) that are not subject to the garbage collection which ensures that the DGC does not interfere with the system timeliness. In this model, developers of the implementation should follow the rule that a reference variable cannot contain a reference to an object which could be released before this reference variable, so that objects in immortal or heap memory cannot contain references to objects in scoped memory (Wellings 2004).

But, using RTSJ memory models introduces the problem of how to implement an identification mechanism into DGC to identify objects in scoped memory area model not referenced remotely.

E- Object Serialization

Distributed schedulability analysis requires the bounding of the message size. In RMI, the message size is defined by both the wiring protocol and the size of serialized objects. The serialization protocol used by RMI allows for non-statically computable classes in case of using open array attributes (de Miguel 2001). Some methods exist to estimate the size of the serialized object (de Miguel 2001), but these methods leave it up to the application developer to ensure that all serialized objects are statically computable.

Also, the authors of (Tejera, Tolosa et al. 2005) referred to the problem that some current implementation of RTSJ of classes that support object serialization, e.g. `ObjectStreamClass`, `ObjectOutputStream`, and `ObjectInputStream`, do not respect the rule of RTSJ memory model that a reference variable cannot contain references to an object which can be released before this reference variable.

3.4.2 RTSJ Support

RTSJ itself still is required to provide solutions for some internal problems. One such memory management problems inherited from RTSJ and affect the implementation of any real time implementation of RMI was mentioned in (Tejera, Tolosa et al. 2005), that RTSJ initializes static object variables in the immortal memory to make them reachable from all memory areas. When the developer tries to place or use these classes in a scoped memory, and there is a reference from the static object to some object in scoped memory, an illegal reference error is thrown.

3.4.3 RMI Tools

The tools used in the development and the running of the RMI-based applications should support the real-time requirements of the middleware as follows.

A- The Registry

Although the registry is a tool used within the framework to look up for remote services, it is basically a remote object. So, just like any remote object it should be modified to work in a real time and a predictable manner.

B- The rmic Compiler

The rmic takes the server objects and generate the proxies (stub and skeleton) required to facilitate the communication. The generated classes do not support any kind of predictability of real-time requirements.

3.4.4 RMI programming model

The programming model of real time RMI needs to make the time required for end-to-end RMI calls predictable, to guarantee timeliness and to avoid unbounded priority inversion. In RMI, the remote invocation is modeled as a synchronous client-server communication mechanism based on a control flow model. For the server, an implementation is free to choose between a single-server invocation thread mechanism (Tejera, Tolosa et al. 2005) or a multi-threaded server model and it may use a load control system by queuing of request and using a thread pool. In order to achieve timely invocation, the RMI programming model and implementation must be able to control the following:

A- The Real Time Parameters of the Involved Threads

The scheduling or release parameters are not considered in operations implementing the remote method invocations at both the client and the server (Tejera, Tolosa et al. 2005). RMI needs to support a different real time scheduling and dispatching mechanisms; this requires flexibility in assigning the different parameters required for each policy (Borg and Wellings 2003).

B- The Network's Real Time Parameters

Of those parameters, i.e. network parameters, current implementations of RMI only allow the exportation of the remote object on a specific port. In general, real time RMI requires to support a broad range of real time networks (e.g. CAN and AFDX) and not to be bound to the unicast classes' implementation of the TCP/IP (Borg and Wellings 2003).

C- Resource Configuration

Resource usage is completely transparent in the current RMI structure. Hence, the RMI is not configurable for QoS. Two main technologies are used in general to configure the network to reserve some of the available resources to determine the required level of QoS:

1. **Integrated Services**; these services control the QoS of a particular session by using packet classification, packet scheduling, policy control, and admission control. It was used by (de Miguel 2001) to support a resource reservation protocol (RSVP) within the RMI transport layer to provide bandwidth reservation facilities that limit the delivery time of sessions of remote invocations, and to provide end-to-end predictability.

2. **Differentiated Services**; this is a packet based priority service used to prioritize the traffic at specific type. It was used by (García 2004) to build an RMI transport layer that allows the application designer to specify the required level of QoS as a vector of minimum bandwidth, maximum delay allowed, and maximum delay bounds allowed. The vector is mapped by the middleware into a marking code that is inserted into the IP packets of the remote invocation to specify a particular per-hop forwarding behaviour on nodes along their path.

D- Failure Semantics

That current programming model of RMI is not concerned with transient communication errors, as it assumes a reliable transparent mechanism using the TCP for implementing its communication wire-level protocol. RMI has exactly *once semantics* in case of absence of failures, and *at most once* semantics in the presence of failures, as in the RMI model; in case of no failure, the data of a single remote method invocation is received at the server side and the call is executed, then the result is sent to the client (call is executed exactly once); otherwise, in case of a failure at the server, this failure can be either before the execution of the required remote method (call is executed zero times), or the failure can be after the execution of the remote method, but before the return of the value to the client (call is executed only one time). These call execution semantics guarantee that the specific instance of access to the remote object is not repeated more than once, enabling one to reason about the safety properties of this remote object. For example, using these semantics guarantees that the invocation of a `withdraw()` method of a remote object, that represents a bank account, is executed at most once time in case of failure, which ensures the safety of this remote object.

These semantics is reflected in the current RMI programming model, RMI throws the `RemoteException` to indicate node or permanent communication failures. This exception is thrown only from the server to the client when the implementation is unable to run the call, or makes the call but detects a failure before the call returned. There is no ability for the server to be informed if a client node fails whilst a server is executing a request invoked by this client (Wellings, Clark et al. 2001).

E- Asynchronously Interrupted Exceptions (AIE)

To be real time, RMI needs to allow appropriate areas for safe interruption and to be able to propagate the exceptions through the network and interrupt the execution on the remote machine. The safe interruption areas are especially important for the network dependent classes not just to enable interruption, but also to ensure that the network is left in a consistent state after handling the exception (Wellings, Clark et al. 2001).

F- Asynchronous Event Handling and Other Services

Using RMI in systems that allows the firing of events and handling those events in remote machines requires the support for asynchronous event handling mechanism. Also, other services may need to be added to the framework such those used in CORBA. For example a service that allows the remote machine to get the propagation type used in the server (i.e. client propagation, server propagated, etc.) (Wellings, Clark et al. 2001).

3.4.5 RMI Implementation

The current implementation of RMI does not consider the real time aspects and they use features and implementation techniques that are not compatible with the specification required for real time Java systems specified in RTSJ (Tejera, Tolosa et al. 2005). Here are some of the problems in the implementation of RMI.

The work in (de Miguel 2001) identified a set of problems that make the *JDK's* implementation of Java-RMI unpredictable, these problems are basically concerning the latencies and blocking times of the communication transport layer of the RMI middleware implementation, the problems are identified and classified into client-side and server side problems as follows:

A- Client-side Problems

RMI uses a connection-oriented mechanism for communication. This mechanism allows the creation of a new connection and hence a new socket for each new invocation and this can lead to using unlimited number of sockets to communicate the same reference with the same server. A proposed solution to this problem is using a session oriented communication instead of the connection oriented communication, where the multiplexing facilities supported by RMI-JRMP protocol is used to enable RMI to create sessions that reuse the same socket for all method invocation in the same reference.

Remote invocation uses the *wait* operation in the synchronized methods that can produce priority inversion. The proposed solution was to use RTSJ's priority inheritance and priority ceiling mechanisms to limit the blocking time of RMI services.

Also, some information such as the socket address is encapsulated in the classes of the transport layer. Adding any new layer, e.g. reservation layer, requires methods to be added to both the transport and the reference layer to access this information.

B- Server-Side Problems

The wire-level protocol, i.e. JRMP, does not include any scheduling information, e.g. priority; thread deadline, remote invocation deadline, into the call stream. So, distributed real-time scheduling concepts cannot be used. A possible solution is to extend the JRMP to include arguments representing the scheduling parameters. However, this solution prevents the communication between real-time and non-real-time RMI implementations. Alternative solution is to extend RMI to identify the server threads priority either by associating a fixed priority to the server, or to allow it to inherit the client priority.

Similar to the client side, the implementation of the server side uses the *wait* operation in the synchronized methods used for accepting the connection and delegates the method invocation. Although, the execution times of these methods are limited, they can pre-empt application threads. As in the client side, using RTSJ's Priority Inheritance and Priority Ceiling mechanisms can solve this problem.

A new connection handler thread is created without any assigned priority for each new received invocation, this leads to an unlimited number of threads without priority. Assigning a maximum number of the allowed created handlers and using a thread pool of them can solve this problem.

Although TCP/IP is the default streaming protocol of the RMI implementation, it is not predictable. Hence, it is not recommended for real time systems. Other protocols based on UDP or ATM can replace it to enhance it (Sun Microsystems Inc 2003). For example authors of (G. Sampemane et al 2006) used Fast Messages (FM) as the underlying transport instead of TCP to minimize the overhead resulting from the marshaling and transport mechanism of the TCP/IP in order to build a high performance Java RMI. Also, as the more efficient user datagram protocol (UDP) cannot be used directly as a streaming protocol in RMI, since it does not guarantee a reliable delivery of invocation request and response messages, a reliable message delivery protocol (R-UDP) was developed based on the UDP protocol in (Krishnaswamy, Walther et al. 1998) to exploit the request-response nature of RMI communications. This protocol (i.e. R-UDP) benefits of the request-response data flow model of RMI where the client sends a single request to the server, followed by the server sending a single reply to the client. Dependent on this model, any explicit acknowledgements used by TCP for requests can be avoided in a reliable protocol (R-UDP) that is aware of the RMI communication. In addition to implementing the R-UDP; the authors in (Krishnaswamy, Walther et al. 1998) extended the RMI framework to support caching. In order to maintain the consistency of the replicated cached objects, a multicasting protocol was implemented in the flexible RMI transport layer to enable the client to propagate updates to all the replicates when required.

3.5 Levels of Integrating RTSJ and RMI

The programming model of Java RMI as a distributed object middleware is based mainly on the control flow of its programming model, in which the execution point, with or without parameters, is moving among application entities (Clark, Jensen et al. 2002).

In order to enhance RMI to support distributed real-time systems, it should maintain the end-to-end timeliness, e.g. time constraints, expected execution time, execution time received thus far, etc., of trans-node application behaviour that is the main defining characteristics of such systems. Where in static distributed systems, these requirements can be instantiated a priori; while in dynamic distributed systems, these properties must be propagated among corresponding computing nodes resource managers in OS, JVM, middleware, etc. (Wellings, Clark et al. 2001).

To achieve the requirement of building a real time RMI that maintains the end-to-end timeliness, In (Wellings, Clark et al. 2001), a three level framework was defined for integrating the RTSJ and RMI, these levels are discussed in the following section.

3.5.1 Level-0 Integration: Minimal Integration

In this level, see Figure 3-6, both the client and the server are implemented as RTSJ objects, but the RMI is used without any modifications. In other words, the remote object implements the `java.rmi.Remote` interface that says nothing about the real time properties of either the server or the underlying transport system. Hence, the stub and skeleton threads, although generated within RTSJ client and server respectively, are generated as ordinary Java objects without any support of real time features and the transport protocol used to implement the connection between the client and the server is not required to be timely. Hence, in this level of integration, the real-time threads at the client can invoke remote methods but they expect no timely delivery of the RMI requests.

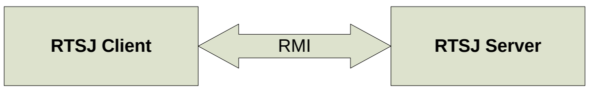


Figure 3-6 Level-0 of Integrating RTSJ and RMI

Although this level of integration has the benefit of not requiring any modifications to the RTSJ or to the RMI middleware, the application programmer is completely responsible for passing explicitly any scheduling or release parameters between the client and the server. Also, it does not provide any enhancements regarding the problems discussed earlier such as the *at most once failure semantic* and the relationships between the client and server clocks.

3.5.2 Level-1 Integration: Real-Time RMI

This level targets implementing RMI in RTSJ in order to achieve the real-time requirements, see Figure 3-7. In other words, all the elements of the RMI framework (stub, skeleton, etc.) need to be modified to be RTSJ components. This modification is applied along the whole three levels of the framework structure:

- **The Programming Model Level;** where real time remote objects implement the `RealtimeRemote` interface instead of the `Remote` interface to be identified remotely.
- **The Implementation Level;** where the transport mechanisms should be modified to allow the propagation of any timing constraints or scheduling parameters between the RTSJ client and the RTSJ server.
- **The Development Tools Level;** development tools, such as rmic, are needed to be enhanced to allow the generation of RMI elements, e.g. proxies, stub, using the RTSJ rather than ordinary Java to facilitate the real-time communication.

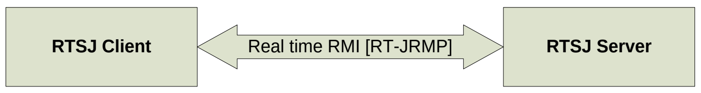


Figure 3-7 Level-1 of integrating RTSJ and RMI

The above proposed modifications lead to a Real-time RMI, in which the proxy at the server side can be viewed as a real-time thread that either:

- Inherits the timeliness parameters from the client if invoked by an RTSJ client.
- Use default timeliness parameters if the client is non-RTSJ client.

Using this level of integration between RTSJ and RMI does not just support the propagation of timing constraints between the client and the server. But also, one more benefit of this level is that it enhances the failure semantics of the RMI. In this level, failures at the client can either be ignored as in ordinary RMI, or the server can be informed by one of the RTSJ asynchronous mechanisms, i.e. firing asynchronous event handler (AE) or throwing asynchronously interrupted exception (AIE). On the other side, this level still offers a set of limitations; these limitations include (Wellings, Clark et al. 2001):

- **Missed Relationship between Real-Time Clocks;** the real-time clocks at both sides are still independent of each other. Also, the serialization of timing constraints as RTSJ timing object requires converting them first into suitable type before serialization, and then reconstructing them after serialization as the RTSJ timing classes are not serializable.
- **Circular Referencing Problem;** it does not introduce any solution to the problems that can result from the circular referencing, i.e. the deadlock and the thread-local data problems.
- **No Remote RTSJ Services;** none of the RTSJ classes define the remote interfaces. Consequently, they can offer no remote services.
- **No Serialization of the RTSJ classes;** most of the RTSJ classes do not implement the serializable interface. Consequently, they cannot be passed across a remote interface.

3.5.3 Level-2 Integration: Distributed Real time threads

As discussed earlier, the implementation of the RMI in real time Java offers a set of limitations; one of the main sources of these limitations is the missing of any system-wide identification of the currently executing thread in the JVMs of the whole nodes in the system. Level-2 of integration, shown in Figure 3-8, aims to solve this problem by using the model of the Distributed Real-Time Threads in which each thread has a unique system-wide identifier.

DRTSJ Client
Distributed Real time RMI
DRTSJ Server

Figure 3-8 Level-2 of Integrating RTSJ and the RMI

The Distributed Real Time Threads model is the core on which the DRTSJ is based. DRTSJ (not finished yet) aims to extend the RTSJ to distributed systems. So, all the components of the RMI framework (stub, skeleton, etc.) need to be modified to be compatible with the DRTS. This modification has to be applied along the whole three models of the framework structure as follows:

- **The Programming Model Level;** distributed real-time remote objects should implement the `DistributedRealtimeRemote` interface for remote identification and then exported by passing them as a parameter to the static method `exportObject()` of the `DistributeRealtimeRemoteObject`, or directly sub-classing them of the `DistributeRealtimeRemoteObject`.
- **The Implementation Level;** changes are needed in the RMI framework; mainly in the transport layer, to facilitate the implementation and accessing of the distributed thread, and allowing the passing of scheduling and state information among DRTSJ platforms, e.g. by extending the RTSJ's RealtimeThread. The required changes are required as well in the JVM itself to support the system-wide nature of the distributed thread.
- **The Development Tools Level;** modify the development tools to generate RMI elements based on the DRTSJ, e.g. the server proxy thread can be distributed real time thread.

A- Distributed Real-time Thread Model

The distributed thread model, (shown in Figure 3-9), was pioneered first in Alpha kernel (Clark, Jensen et al. 1992) and Mk7 (Wells 1994) as a part of those distributed operating system. Then, used on middleware level (e.g. Real-Time CORBA(Object Management Group (OMG) 2005)) and DRTSJ aims to use it as part of the Java programming language to support the real-time RMI middleware (Clark, Jensen et al. 2002).

The model of distributed real-time thread abstracts the control flow of the distributed object systems. It executes remote methods, like a local one, directly itself by extending and retracting itself between objects and (transparently) nodes. In other words, the thread's locus of control can move freely across the distributed system by calling methods in remote objects [74, 81-83].

In the whole distributed system, there may exist one or more concurrent distributed thread, see Figure 3-9, where each distributed thread has some defined distinct features as follows (Anderson and Jensen 2006):

- It has a unique system-wide identifier.
- It has only one *root* that is the site from which the distributed thread originated.
- It has exactly one execution point called the head at any point of time.
- Other sites that are hosting a part of the distributed thread are called segments sites.
- The end-to-end timeliness of the sequential execution of methods along its segments makes the distributed thread the entity to be scheduled.
- It uses a model of client propagated release and scheduling parameters to carry its scheduling parameters as it transits node boundaries.
- Some of its internal segments may have its own timeliness (scheduling segments) rather than the whole end-to-end timeliness.
- The internal segments may be composed of other nested segments that also can have their own timeliness.
- Control flow can be forked by creating or awaking other distributed threads explicitly or implicitly. The newly forked distributed thread may be explicitly created as two-way (synchronous) invocation, which means that the original thread waits for it to finish its execution; or be implicitly created by a one-way invocation that is executing in parallel with the original thread.
- It define several categories of scheduling events including:

Actions on Scheduling Segments. Such as entering, exiting, or updating a scheduling parameter.
Actions on Resources. Such as requesting or releasing a managed resource.
Actions on Nodes. Such as entering or exiting a node.

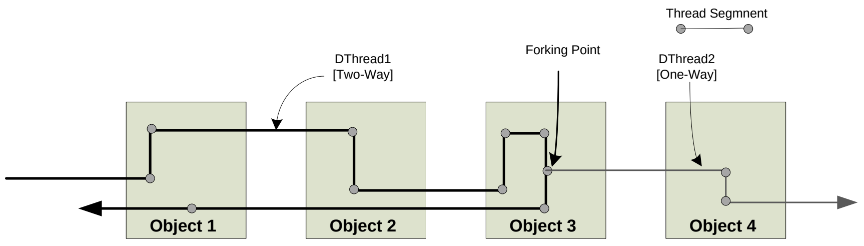


Figure 3-9 he Distributed Thread

For a distributed thread in distributed real time system a set of basic methods can be defined, these methods are classified into two main groups (Wellings, Clark et al. 2001):

- **Manage Execution State Methods;** this group includes methods to start and interrupt the distributed thread even on a remote site.
- **Schedulability Management Methods;** in order to enable operations like mode changes, a set of methods are assumed to set and get the release and scheduling parameters of the running distributed thread at any time and at any node.

B- Requirements to Support Level-2

In (Wellings, Clark et al. 2001) the requirements of the RTSJ and JVM to support level-2 was investigated and summarized in the following:

- The remote invocation of the *schedulability management* methods requires enabling the release and scheduling parameters to be serialized and or accessed remotely across node boundaries. However, RTSJ's classes of those parameters, e.g. `ReleaseParameters`, and `SchedulingParameters`, do implement neither the `Remote` interface nor the serialization as stated before. Hence, RTSJ classes still need modification or extension of these classes to support the distributed real time thread model.
- Also, event handlers in RTSJ may be bound to a thread either permanently or at run-time, but it is not clear how many handlers are bound to each schedulable thread. In DRTSJ, the proposed definition of distributed events and their handlers interfaces assume that they are defined as distributed remote objects that for:

1- **Distributed Events;** extends the `RemoteAsynchronousEvent` interface that supports the methods for remote firing of an event and attaching a remote handler to it.
Distributed Event Handler; extends the `GlobalAsyncEventHandler` interface that supports only the method `getSchedular()` needed by the remote node to determine which scheduler to be informed when an event occurs.

- The attaching of a remote handler to an event is not allowed. This is because any distributed event should not be serializable; otherwise, interrupts responsible of its occurrence should be passed across the network. Also, handlers should not be serializable; as if it is serializable, it would be confusing and unclear where the handler should be scheduled.

- The level-2 model assumes the hosting of the distributed thread within a cluster of nodes. So, DRTSJ is responsible to provide and coordinate a cluster-wide clock and synchronize it to some delta and within a defined accuracy of UTC.

- Failure responses and semantics resulting from a segment crash failures can be coordinated by DRTSJ according to the location of the failure into three types as follows:

1- **Failure at Head;** a remote exception is thrown to the site hosting the previous segment.

2- **Failure at Internal Segment (Non-Head-Non-Origin);** there are two semantics:

   a. The head site either ignores, or throws an AIE or fire an AE).
   b. DRTSJ throws exception in the segment site previous to the failed one if it tried to access the failed segment

3- **Failure at Origin;** the same as part (a.) stated in type 2.

- Applying the ATCs to distributed threads adds extra complexity as the fired AIE can be generated at any segment but must be propagated to the moving head segment. Furthermore, there may be outstanding AIEs for the thread. So, two models are assumed for the implementation:

1- All outstanding AIEs need to be stored at the origin to be checked each time the distributed thread is interrupted to see if AIE should be thrown.
2- All outstanding AIEs have to be carried with the head of the thread.

3.6 RT-RMI (Level-1) Models

This section presents two models proposed in (Tejera, Tolosa et al. 2005) for implementing the second level (Level-1) RT-RMI in RTSJ as a part of their high integrity Java project (High Integrity Java Application Project):

RMI-HRT. This model was built for safety critical systems. It is based on the Ravenscar Java proposed by the authors of (Kwon, Wellings et al. 2002). In Ravenscar Java, the features with overhead and complex semantics; which may prevent the predictability, are eliminated. Two execution phases are defined in Ravenscar Java:

a. **Initialization Phase;** in which the non-real time tasks are carried out, e.g. creation and exportation of the remote objects, pre-load the required classes, etc.
b. **Mission Phase**; where the application's real time tasks are executed.

RMI-QoS. This mode target business-critical systems; i.e. soft real-time. This model aims to build a communication model that provides a service with a minimum required quality by reserving CPU, memory, threads and bandwidth to guarantee enough resources for that goal. The phases of this model in sequence are as follows:

a. **Server Initialization;** the server is initialized to be ready to accept sessions from the client.
b. **Reference Phase;** the client gets a reference to target remote object, and establishes a session with the server.
c. **Negotiation Processes;** clients try to negotiate and manage reservations as best effort invocations over the sessions.
d. **Data Transactions**; once a reservation is made, clients can make remote invocations to the server with QoS guarantees.

In the RMI-QoS, the phases, except the server initialization, may occur at any time during the application lifetime to support the dynamic nature of this model.

The authors of (Tejera, Tolosa et al. 2005) identified the approaches of the adaptation of RMI to support the Level-1 integration with RTSJ in their proposed two models. These adaptations are summarized here for both models.

3.6.1 Adaptations in the RMI-HRT

The RMI-HRT targets safety critical systems, so its model has many restrictions that are required for such systems as explained in the following.

A- Thread Concurrency Model

In the RMI-HRT, the researchers adopted the model presented in (Borg and Wellings 2003), where at the server, the schedulable threads of the process are separated into two main parts; acceptor threads and handler threads. The acceptor listens to the incoming invocations and once received, a suitable handler is chosen from the pool and assigned to manage the invocation according to the scheduling information indicated by the client’s invocation, and the acceptor starts to wait for other invocations. Once the method is executed and the return parameters are sent, the handler returns to the pool.

On the client side, the remote invocation is done transparently through a stub, which carries all the necessary parameters including the scheduling parameters, and propagates them, with the invoked method, through the net. Also, the stub waits for the return values by listening to a port that can be specified by the developer. Hence each invocation will listen to a different port.

According to Ravenscar Java, acceptors and handlers (which are schedulable objects), should be generated at the initialization phase. Consequently, a set of acceptors is created for each exported object to handle different clients, where all the handlers are kept in a pool of handlers. There may be one pool of handlers for each exported object or just one pool of handlers shared by several exported objects. To attend all the invocations, it is enough that the server has for each client, one handler for each method invoked with different scheduling parameters (Tejera, Tolosa et al. 2005).

B- Programming Model and Specification

The following changes have been proposed in the programming model of the RMI-HRT:

1- The use of `RMIClassLoader` and `RMISecurityManager` should be restricted to the initialization phase only.

2- The support of the propagation of scheduling parameters can be through the re-implementation of the `invoke()` method to include a field that represents them.

3- The `RTRemoteStub`, `RTRemoteServer` classes and interface `RTRemote` are created instead of the original ones, to support the real time semantics of the remote objects as proposed in (Borg and Wellings 2003).

4- *rmic* must be modified to manage the stub that should implement the `RTRemote` and predictably manage the memory.

5- An interface to act as the abstract of the `ConnectionHandlers`' pool is required to include operations such as creating it. Also, operations to support the binding of a remote object dynamically to a certain `ConnectionHandlers'` pool is needed, e.g. `setHandlerPool().`

6- The constructors and methods of `UnicastRTRemoteObject` are updated to support acceptors, scheduling parameters beside the net and port parameters.

7- The serialized objects should implement the `RTSerializable` interface that allows computing the network resources required in the transmission of an object, and the maximum execution time needed for serialization.

8- For safety, networks that provide mechanisms to guarantee the message delivery on time (e.g. CAN, AFDX, etc.) should be used instead of the TCP that is used by default in RMI.

C- Memory Model

As the RMI- HRT follows the Ravenscar Java profile in order to simplify the design and implementation of virtual machine and eliminating sources of indeterminism, it does have some restrictions on memory management. It does not allow for nesting memories or sharing different scoped memory areas by different schedulable objects. Two approaches to achieve this in both client and server implementations are as follows:

I-Using Immortal Memory

At the server side, if arguments of the remote method are immortal, they can be referenced by other objects in immortal memory or by new objects in scoped memory. Hence, the handler of such remote method must exit the scoped memory first before calling the corresponding method. However, the server's temporary objects must be created in scoped memory, otherwise they will never be reclaimed; this requires the developer to enter the scoped memory before creating any temporary object. At the client, temporary objects created by the stub can be created as immortal memory and reused in each invocation. This approach is simpler for the developer to avoid nested scopes but requires more immortal memory.

II-Using Scoped Memory

At the server side, if arguments of the remote method are in a scoped memory area, the developer should avoid to make references from objects that are in immortal memory. Also, to avoid nesting of scoped memory, it is not allowed to enter a new scoped memory area as the handler is already in one (where the arguments are).

At the client, the stub can enter scoped memory at the beginning of execution and exits when finished. So, the created temporary objects are reclaimed. However, this requires the invoking thread to be out of any scoped memory at the moment of invocation to avoid generating nested scoped memory. A variation of this is to consider that the calling thread is itself within a scoped memory and knows the required memory consumption needed for the stub. Hence, the temporary objects are created in this scope and eliminated after the calling thread destroys that memory. This approach requires more effort from the developer but requires less immortal memory.

3.6.2 Adaptations in the RMI-QoS

This model is a resources-reservation based model and it targets the business-critical systems, so it has a less degree of the constraints as explained in the following.

A- Thread Concurrency Model

In this model, the client negotiates the server for a reservation of its resources, once the reservation is admitted it needs to be kept along all the remote invocations from the client to the server. In order to support this requirement, this model enforces the session-oriented approach to be used instead of the RMI default connection-oriented approach. Therefore, once a connection is made between a client and a remote server, any subsequent invocations will use the same connection. In this model, the server has the following threads for every remote object:

- **Listener Thread**. Waiting for incoming requests from clients to the remote object. Once the listener accepts a session with a client, it bounds an acceptor thread to this session.
- **Acceptor Thread**. To hand out negotiation requests, best-effort, and guaranteed remote invocations to the associated handlers to execute it.
- **Handler.** To carry out negotiation processes, best-effort, and guaranteed remote invocations. Where any negotiation or remote invocation sent before reservation is carried out as best effort. Once a reservation is admitted, the remote method invocation is carried out with QoS guarantees.

B- Programming Model and Specification

The following adaptations have been proposed for this model:

- The interface `RTResRemote` is introduced to identify the remote object with QoS facilities based on the resource reservation.
- Classes `RTResRemoteStub`, `RTResUnicastRemoteObject` are assumed to provide negotiation methods needed before building the session.
- `RTResRemote` methods manage dynamic negotiation and reservation of resources (CPU, memory and bandwidth) as well as sessions.
- Methods for reservation management (create, modify and delete reservations) take into account the following parameters:

1. The required level of thread concurrency at the server.
2. If the negotiation is to a specific method or to a group of methods.
3. The rate of the invocations and the execution time budget.

- A new Class `Reservation` is defined to keep track of the reservations made and available resources. This class provides *get* and *set* methods to access this information.
- The *rmic* and are adapted as in HRT-RMI.

C- Memory Model

Memory management proposed in the RMI-QoS requires that threads with application lifetime, e.g. Listener thread, are to be created during the initialization phase in immortal memory. However other objects, e.g. acceptor and handlers, are allowed to be created and destroyed dynamically at any phase in any scoped memory, to avoid consuming the immortal memory. Also, using and creating nested scoped memory areas is allowed at any phase in this model. This enables the stubs to execute in a nested scoped memory area to make sure that marshaling and unmarshaling processes do not consume memory.

A memory management model based on the memory model of the RTSJ was presented in (Basanta-Val, García-Valls et al. 2004; Basanta-Val, Garcia-Valls et al. 2005) for the RMI-QoS. The model was proposed for simple distributed systems that keep their internal state and create well-known number of objects of a known size per remote invocation. This model includes two scope stacks as follows, see Figure 3-10:

- **Client Side Scope Stack;** it uses the default allocation context of the invoking thread to allocate the object that returns as a result of the remote invocation.
- **Server Side Scope Stack;** it is more complex and the authors defined two structures of it according to the situation at the server:

1. *The scopes stack when a remote object is created.* The scopes stack of the thread that creates the remote object on the server.
2. *The scopes stack when a remote object is invoked.* The scopes stack of the handler thread.

The scope stack of the allocated memory at the server in this model is fragmented into two memory contexts as follows:

- **Creation Context;** it stores the state of the remote object, i.e. the remote object instance and the objects referenced by its attributes. Objects in this context are never destroyed before the remote object is destroyed. Hence, it contains a set of scoped memory instances and immortal memory. Where the scoped memory in this context is maintained along the lifetime of the object by incrementing their counter once the object is exported, and decrement the counter once the object is un-exported. The memory required for this context has to be either pre-allocated during the creation of the remote object or taken from shared pool of objects.

- **Invocation Context;** it stores the temporary objects of the parameters of the remote invocation. Hence, it contains a set of scoped memory instances only, as objects created within this context are destroyed after the remote invocation ends. The amount of memory needed in this context has to be finite and bounded. An internal counter is used for each invocation context, when the counter changes from one to zero, the objects that it contains are destroyed and the memory of the invocation context is reclaimed.

A new assignment rule; the No-heap Remote assignments rule (NhRo); was assumed in this model to extend the RTSJ’s assignment rule. This rule is stated as follows:

> *"The NhRo rule forbids the references from objects stored in the creation context to objects stored in the invocation context. The opposite is possible: objects allocated in the invocation context may reference objects allocated in the creation context. "*

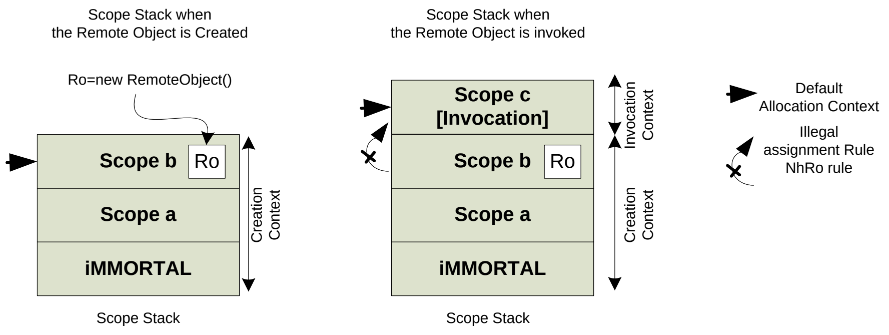


Figure 3-10 RMI Memory Management Model for RMI-QoS

To support the assumed memory model, the authors assumed an extended RMI middleware structure. The key elements of the server side structure are:

- **Remote Object Table;** contains the information of the remote objects, e.g. remote object identifier, reference, and creation context of the remote object.
- **Thread Pool;** to supply the schedulable handler threads used at server side during each remote invocation.
- **Memory Area Pool;** provides a scoped memory for each invocation context and recycles it after the remote invocation. The memory of the memory area pool is pre-allocated as a linked list and configured by two parameters; the maximum number of invocation context and its size.

During the remote invocation, the proposed extended structure at the server side works as follows**:**

- The handler thread reads the `ObjID` sent by the client; then
- The handler uses the `ObjID` to query the *Remote Object Table* to get the creation context and push it to its stack.
- Then, it allocates the memory required for the invocation and pushes it on top of the stack.
- Then, it allocates the parameters of the remote invocation in the invocation context, and executes the required method of the remote object.
- Finally, it sends the result of the invoked method to the client and pops the invocation context and the creation context.

3.7 RT-CORBA

CORBA is a language independent framework standardized by the OMG to support distributed object computing. CORBA is a distributed object framework based on the ORB design pattern, where the ORB is the central component that is responsible of finding the required object implementation, transparently activating it if necessary, delivering the requested invocation to it and finally returning the result to the client. The Real time CORBA specification (Object Management Group (OMG) 2005) was presented by the OMG group to support the QoS needs of distributed real time systems within CORBA architecture by providing extensions that identifies capabilities that ORB end systems must integrate and manage both *vertically* from network interface to application layer, and *horizontally* from peer to peer (Schmidt and Kuhns 2000).

3.7.1 RT-CORBA Features

RT-CORBA, see Figure 3-11, improves the distributed systems predictability by bounding the priority inversions and managing system resources end-to-end through standard features that allow configuring and controlling the following system resources (Object Management Group (OMG) 2005):

1- **Processor Resources.** RT-CORBA offers the following configurable properties and mechanisms for managing the processor resources:

a. **Thread pools;** it supports using thread pools of maximum allocation size and with the ability of partitioning it into lanes of certain priority for each lane.
b. **Priority Mechanisms;** it supports both client propagated, and server declared models and it uses the CORBA priority as a unified priority for all objects,

and provides mapping mechanisms to map it to the native priority at each node.

c. **Intra-Process Mutexes;** it offers a global mutex interface to ensure semantic consistency between CORBA applications and internal synchronization mechanisms used by the ORB.

d. **Global Scheduling Services**; it supports both fixed and dynamic real time scheduling services.

2- **Communication Resources.** It enables the selection and the configuration of protocol policies. Also, allows the explicit bindings to server objects using priority bands and private connections.

3- **Memory Resources.** It buffer requests in queues until a thread of the thread lane is free to handle it. Also, it bounds the size of thread pools by specifying a maximum limit for it.

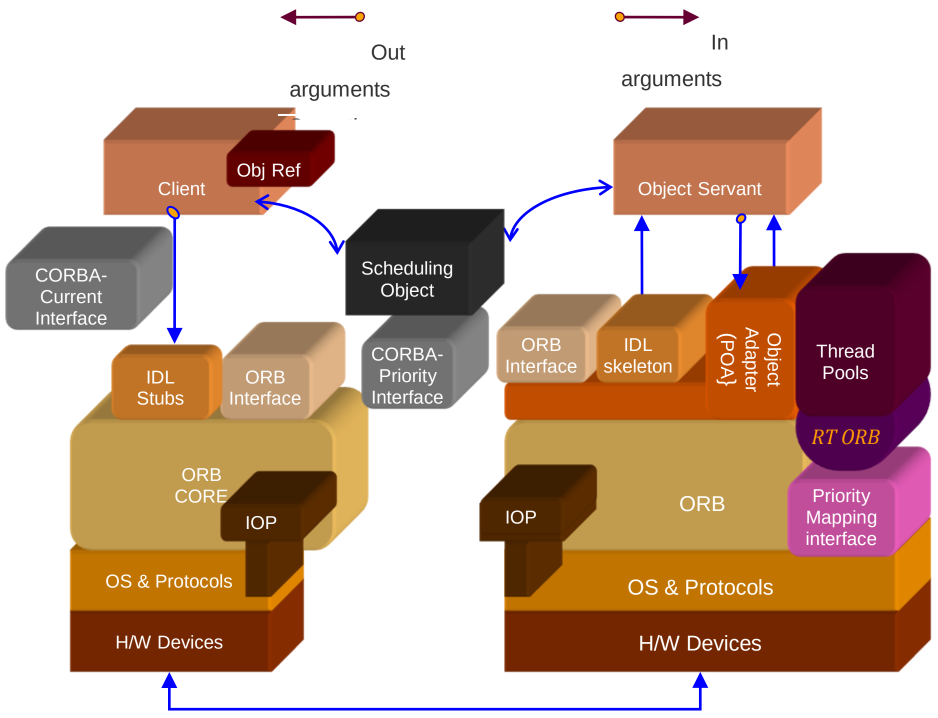


Figure 3-11 Real-Time CORBA Architecture

3.7.2 RT-CORBA and the RTSJ

RT-CORBA was adopted several years before the RTSJ was standardized. Hence, the Java mapping of the original RT-CORBA does not use any of the RTSJ features (Krishna, Schmidt et al. 2004). Some work has been made to implement the RT-CORBA using the RTSJ features. For example, RTZEN is an open source that uses an RTSJ-based implementation of the RT-CORBA's ORB (Raman, Zhang et al. 2005). The RTZEN targets the need to have a Java-based predictable implementation of real time CORBA by taking advantages of the RTSJ features in order to eliminate the unpredictability caused by the GC, and improper support of thread scheduling, and to ensure predictability through the use of appropriate data structures, threading models, and memory scopes (Raman, Zhang et al. 2005).

A- Architecture of the RT-ZEN

The original architecture of ZEN was an optimized Java implementation of the RT-CORBA built, where the key ORB components involved in request/response processing, i.e. acceptors, connectors, transports, and thread pools; are originally allocated in the heap, without any consideration to real time Java (Krishna, Schmidt et al. 2003). In order to build the RT-ZEN, the ZEN's client-request processing steps was analyzed in (Krishna, Schmidt et al. 2004) and it is found that it spans the following layers:

- **I/O Layer**; including acceptor-connector and reactor.
- **ORB Core Layer**; GIOP message parsers, CDR streams and buffer allocators.
- **Object Adapter Layer**; thread pools and the POA.

Along these layers, the request/reply processing is a thread bounded repetitive task that is independently processed, i.e. two requests do not have any context. Also, each request has a set of allocated objects that remain valid only for one cycle. Therefore, the RTSJ features can be applied in RT-ZEN using the following three strategies (Krishna A.S. and Schmidt D. C. 2004):

- **Real-Time Threads**; the logic of each thread-bounded component, e.g. Acceptor, Transport, Connector, are executed using real-time threads, where each real-time thread is associated with a scoped memory region as the current allocation context.
- **Scoped Memory**; as objects in the up-call processing are valid only for one cycle, it is associated with scoped memory regions that enable reclaiming memory safely after finishing each cycle of processing, thus minimizing the number of garbage collection sweeps. Creation of scoped memory regions requires the size of the memory region to be specified. However, the request/response processing in ZEN is dynamic. Hence, RT-ZEN uses nested scopes for each response/de-multiplexing phase resulting in a set of memory spaces, shown in Figure 3-12, as explained below (Krishna A.S. and Schmidt D. C. 2004).

1. **I/O Space;** the inner logic class of each thread bound component is created within a scoped memory $m_{I/O}$. So, during ORB execution, multiple clients can connect to it,

creating transports with dedicated $m_{I/O}$ for every active client. These regions are collectively referred to as I/O space.

2. **ORB Space;** for each new data event received by the transport, a buffer is created to hold the message and an appropriate message parser is associated to it. Both the request buffer and message parser are created in a new nested memory region called m_{ORB}.
3. **POA Space;** once the parser detects the appropriate POA and the servant, a new object is created to hold the up-call information on the skeleton and a worker thread in the thread pool performs the up-call. Also, a CDR buffer is created to hold the response that will be sent to the client. Both the up-call objects and the output buffers are created in a new nested scoped memory region m_{POA}.

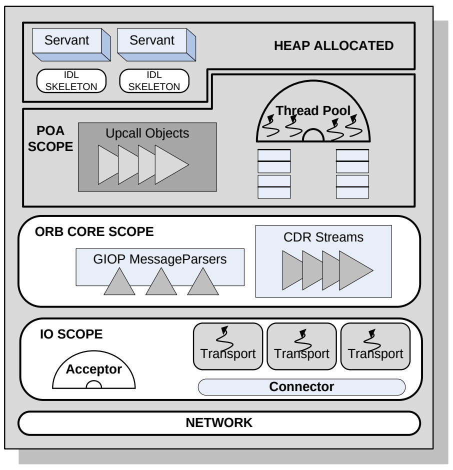


Figure 3-12 Scoped Memory Application- ORB Internal View

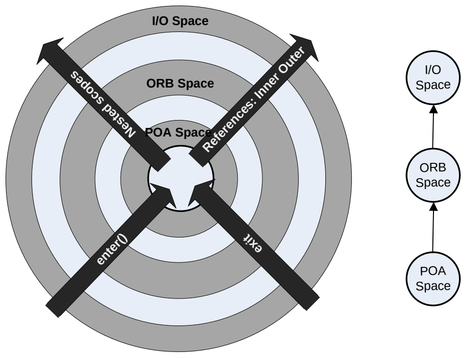


Figure 3-13 Scope Nesting the ORB of RTZEN

The Scope stack of structure above nested regions within the ZEN ORB core shown in Figure 3-13 is typically an extension of the RTSJ scope stack structure, where Memory regions are entered from the outermost to the inner most, while the references are allowed only from the inner most to the outer most. On completion of the request, the memory is regions are exited from the inner most to the outer most (Krishna, Raman et al. 2003).

B- Support RTSJ Features via CORBA Policies.

One of the main goals of RT-ZEN is to incorporate the RTSJ features within the ORB core and POA layers without requiring any modifications to the RT-CORBA specifications. However, it is important to allow the RTSJ developers to enhance the predictability of their CORBA applications using RTSJ aware features. This can be achieved by creating custom policies in the ORB, e.g. at the POA level(Krishna, Raman et al. 2003). Two possible policies are assumed in (Krishna A.S. and Schmidt D. C. 2004):

- The Type of RTSJ Memory Region; the POA conceptually is responsible for managing the lifecycle of application defined servants that are generally are heap allocated. But for servants implemented in RTSJ, the developers may create it in non-heap regions to enhance predictability. In this case, it is not possible to register these RTSJ servants with non-RTSJ aware POA. Hence, for RTSJ servants, a policy can be used to specify the allocation of memory used in POA to be RTSJ aware.

- The Type of Real Time Thread Policy; in the design of ZEN architecture, the `NoHeapRealTimeThread` cannot be used for request processing, as the application layer is allocated in the heap memory to be compatible with RT-CORBA, which is language independent. Hence, in case of RTSJ aware server application, a policy at the POA level can be used to customize the type of the real time thread used in the request processing to be `NoHeapRealTimethread` to enhance predictability.

3.8 Summary

In this chapter, we aimed to discuss the recent research efforts made on Java to support building real-time middleware solutions. So, the RMI, CORBA technologies in Java that support the distributed object paradigm of middleware were presented from a real-time perspective. The general structure of each of them was presented, and how these structures do not support the real-time requirement were discussed; also, the recent research to enhance these structures using the new facilities of the emerging real-time Java, like scoped memory and distributed threads, was presented. We have seen that using the RTSJ is still in the early stage of developing real-time middleware, and there is no RTSJ-based commercial middleware implementation in the market yet, as there is still a lot of work to be done on in order to get the RTSJ-based middleware solutions to reality. Many of these problems are related to the difficulties of using the new memory model presented in the RTSJ, especially that most of the current Java libraries cannot be used directly in the implementations, as they are not restricted to the rules of this new memory model. Hence, there is a need to build new elements that respect the rules of the RTSJ, and in the same time support the basic elements required for the middleware solutions. We propose that these software elements can be built as RTSJ-based software reusable components, which can be integrated together to form the required middleware, where these components are built using design patterns that have to respect the RTSJ memory rules, and have to have a predictable timings of execution. So, in the next chapter, we first present the basic concepts of RTSJ, as it is the core over which the proposed patterns and components can be built. Then we provide a quick overview of the design patterns and components technologies as two commonly used software mechanisms that are used in building many middleware solutions, real-time and non-real time in order to understand the challenges that are faced when using them in the real-time domain in general, as well as discussing the research work done on them to be used in RTSJ-based applications.

Chapter 4

Patterns and Components *for* Real-time Java Distributed Systems

In the previous chapter, we discussed the recent research made to build real-time middleware solutions, where we discussed the research toward developing real-time implementations of the RMI and CORBA models. Most of these implementations are based on using the new features of the RTSJ to build the inner structure of these models. So, in this chapter, we aim to provide a deeper view of the new features of the RTSJ that make it more appropriate for building real-time middleware applications. Then, as we aim to build our real-time middleware model based on the component technology, we provide an overview of the principles of the software components technology. In addition to that, we provide an overview of the software patterns technology, which represents a cornerstone of building our proposed component model presented in the next chapter.

4.1 Real-time Java

The Real-Time Specification for Java (RTSJ) (G. Bollella, B. Brosgol et al. 2006) was formalized in June 2000, and it aims to support the writing of real-time code from a different direction used by other software development platforms; e.g. real-time operating system application programming interface, as it aims to enable the developer to write a portable, real-time, high-level Java code. In order to achieve this goal, the RTSJ embraces the notion of real-time scheduling theory as a fundamental principle for the development of applications that have temporal correctness requirement (Eric J. Bruno and Bollella 2009).

An important goal of developing the real-time Java is to provide a general purpose Java implementation for use in real-time environments, such as servers and embedded systems without precluding its implementation to any of the existing Java environments. So, for developing the RTSJ, the authors assumed that the Java syntax should be used without any additions or changes to the language syntax. Also, to ensure compatibility, it was required that the current non-real-time Java programs should run on the virtual machines that support the RTSJ. However, as the predictable Java application execution is the primary concern of the RTSJ, tradeoffs in the area of general purpose computing can be made where necessary.

Another goal of the real-time Java is the support of features of real-time systems, such as resource budgets, reservation for CPU time, and managing the memory allocation rate and the total memory usage, as well as the ability to specify the real-time constraints within the application code. Also, it has to be flexible in the degree of the real-time support and associated resource management, e.g. offering support for both soft real-time and hard read-time systems.

4.2 Real-time Java Enhancements

In this section, we summarize the important enhancements outlined by the RTSJ for supporting real-time programming in the RTSJ.

4.2.1 Threads and Scheduling

Non-real-time environments are not concerned with the details of scheduling, but in real-time systems, the scheduling must be precise as the main concerns of real-time programming is to ensure the timely or predictable execution of sequences of machine instructions, where in general, various scheduling schemes name these sequences of instructions differently. Examples of the names of these sequences of instructions include threads, tasks, modules, and blocks. The RTSJ introduces the concept of a *schedulable object* to represent the element of the language that its scheduling and dispatching is managed by the instance of the scheduler to which it holds a reference (G. Bollella, B. Brosgol et al. 2006).

The schedulable object is any object that is instantiated from a class that extends the `schedulable` interface, which itself extends the `Runnable` interface. In RTSJ, there are four basic classes that implement the schedulable interface, see Figure 4-1.

RealtimeThread (RTT). This class represents a real-time thread that runs by default in the heap; however, it is possible to initialize it in any other RTSJ memory area defined by the RTSJ by specifying the required memory area in its constructor. During its lifetime, it can move to any other memory area without restrictions.

NoHeapRealtimeThread (NHRT). It is a type of RTT with the additional semantic that it cannot access the heap in any way. So, it has to be initiated either in any one of the scoped memory areas or in the immortal memory (both are described later in this chapter). Once, it is started it has to continue execution through its lifetime in scoped memory area(s) and/or in the immortal memory, and its code must never access or reference an object on the heap otherwise, a `MemoryAccesError` exception will be thrown.

AsyncEventHandler (AEH). As many real-time systems are event driven, the RTSJ introduces the asynchronous event handler to encapsulate code that is to be released after an instance of an `AsyncEvent`, which is a class that represents an event. Internally in the JVM, one or more server real-time threads can be used to handle several AEHs as, in general, the AEH is assumed to be a task that is not frequently occurring and there might not be a need for dedicating a real-time thread to handle it individually.

BoundAsyncEventHandler (BAEH). A bound asynchronous event handler is an instance of the `AsyncEventHandler` that is permanently bound to a dedicated real-time thread. Bound asynchronous event handlers are useful in situations where the added timeliness does worth the overhead of dedicating an individual real-time thread to the handler. An individual server real-time thread can only be dedicated to a single bound event handler.

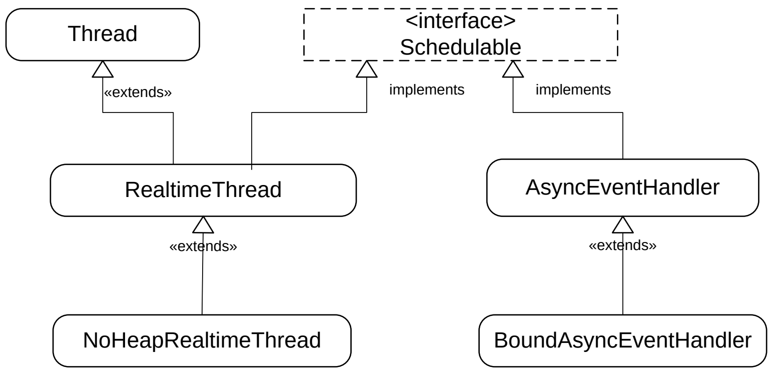


Figure 4-1 Class diagram of RTSJ's schedulable objects

Instances of the above classes have to be managed by a scheduler. Although it permits the use of other schedulers, the RTSJ supports a real-time scheduler that uses the fixed-priority preemptive scheduling mechanism, this scheduler extends the abstract class `Scheduler` and it requires the underlying JVM to support at least 28 real-time priority levels in addition to the 10 priorities called for by the normal JVM.

The RTSJ requires a number of classes, *parameters classes*, that defines the temporal information required to support the execution of the above schedulable objects. An instance of any of these parameters classes should hold a particular resource demand characteristics for one or more schedulable objects. For example `PriorityParameters` sub-class of the `SchedulingParameters` class contains the execution eligibility metric of the base scheduler, i.e. priority. So, when it is assigned to a certain schedulable object, it is used by the scheduler to schedule the execution of this schedulable object amongst the other schedulable objects in the system. Another important example of these parameters classes is the `ReleaseParameters` class, which defines exactly the timing characteristics of starting the execution of the code attached to a certain schedulable object. A final example of these parameters classes is the `MemoryParameters` which defines the memory context in which the schedulable object will run in and its characteristics; e.g. initial memory size, maximum size, types of allocation rate, etc.

4.2.2 Memory Management in the RTSJ

Java adopts an automatic memory management scheme that uses garbage collection algorithms to optimize the use of the heap memory offered by the JVM. However, garbage collected memory heaps have always been considered an obstacle to real-time programming, due to the unpredictable latencies introduced by the garbage collector (G. Bollella, B. Brosgol et al. 2006). Therefore, the authors of the RTSJ sought two directions; the first direction is to allow the use of a garbage collector that allows as much as possible for the job of the garbage collection to not intrude on the programming task and it gives the developer the ability to reason about its effect on the execution time, preemption, and dispatching of real-time threads. For this direction, as the RTSJ authors know that multiple garbage collection algorithms exist, the RTSJ does not specify or endorse any garbage collection algorithm. The other direction is to avoid the use of the garbage collector completely by allowing the allocation of objects in other memory areas other than the heap, where objects in these memory areas are not collectable by the garbage collector. In the following sections we will focus more on these memory areas introduced by the RTSJ as well as the rules and constraints of using them.

A- Memory Areas in the RTSJ

The RTSJ introduces the concept of a memory area. A memory area represents an area of memory that can be used for allocating objects. The idea behind defining these memory areas is to add the support of different memory management solutions that have higher predictability, by defining restrictions on what the garbage collector and the system can do on objects allocated within these memory areas. RTSJ defines four different memory area types that are shown in Figure 4-2. In this section we discuss the features of these different memory areas.

The Heap

The heap memory is referenced by the singleton Java object, `HeapMemory`. As it was in the standard Java, the heap represents the area of the free memory used for the dynamic allocation and automatic reclamation of Java objects. The garbage collector is the main player for the dynamic memory management of the heap. However, the heap, due to the unpredictability of the garbage collector, may not be used by schedulable objects that have hard real-time requirements. So, in order to have predictable memory management for objects created and accessed in the heap, it is important to present a real-time garbage collector as stated before.

The Scoped Memory

The scoped memory is introduced in the RTSJ as a new concept for memory management where it represents a temporary memory that is not using garbage collection algorithms. The scoped memory areas are represented by the abstract `ScopedMemory` class or one of its subclasses shown in Figure 4-2, where instances of these subclasses are created at run time dynamically to represent memory areas regions with certain parameters such as the initial size, the maximum size that it can grow to. The main memory management feature of the scoped memory areas in RTSJ is that objects created in it are not garbage collected like those objects that are created in the heap. Instead it uses the reference counting mechanism to discard the full contents of the scoped memory area at once when all the schedulable objects that are entered it or started running in it terminates their execution in it. At the point of discarding all the objects in the scoped memory area, the finalized method of all these discarded objects is executed, although no garbage collection occurs in the scoped memory area. It is shown in Figure 4-2 that the RTSJ defines two main subclasses for the scoped memory areas; the `LTMemoryArea` that represents a scoped memory area with linear-time allocation, and `VTMemoryArea` to represent a scoped memory area with variable time allocation. As seen from its features, the scoped memory area type has high predictability, so it can be used by tasks that use schedulable objects with either hard or soft real-time requirements.

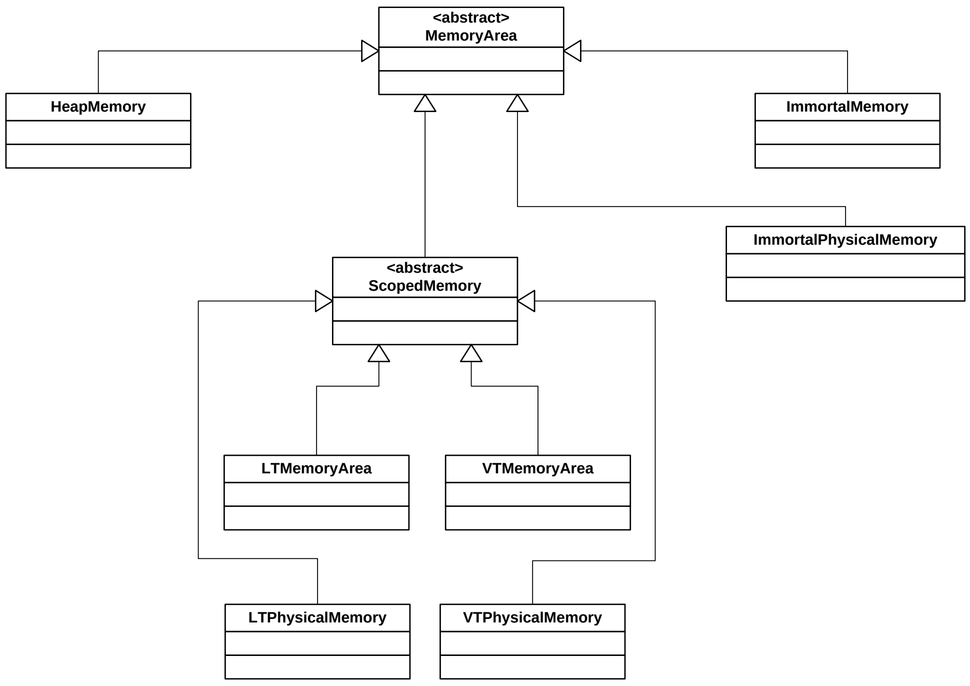


Figure 4-2 Classes of Memory Areas in RTSJ

The Immortal Memory

The immortal memory is represented in the RTSJ by the singleton class `ImmortalMemory`. The immortal memory from one point of view is like the heap in that there is only a single immortal memory region and it is created when the real-time JVM starts up. The difference between the immortal memory and the Heap is in the way of managing the lifetime of the objects created within it. In the immortal memory, the lifetime of the objects created in it, is the life of the JVM, i.e. objects created in it are never deleted before the application termination. Hence, the garbage collector is not responsible for managing this memory area and it is the responsibility of the programmer to optimize the management of the object allocation within it, to not overflow its capacity. By default, all static objects are created in the immortal memory, as are interned String objects, and static initializes execute there. The programmer is also allowed to create any objects within this memory area without restriction, so it can be used as an allocation context for any schedulable objects.

The Physical Memory

The use of physical memory as an allocation context may not be needed by many real-time Java developers, as it will be needed when the application has to communicate through certain hardware to the outside world, e.g. robotics applications. So, RTSJ defines the classes that can be used to create memory areas within a specified range of physical memory. These classes are `LTPhysicalMemory`, `VTPhysicalMemory`, and `ImmortalPhysicalMemory`.

B- Rules of using Scoped Memory as an Allocation Context

A certain instance object that represents a scoped memory area can be used as an allocation context for a schedulable object using one of the following three methods:

1- By assigning it as the memory area parameter of its constructor, so that the schedulable object will consider it as its allocation context once it starts execution.

2- By using the method `enter()` of this memory area instance object to change the allocation context from the current one to be this instance. The use of this method creates a nested scope within the current scope for the schedulable object that executes it. So, as this method can be used several times by the same schedulable object to change the allocation context among several scoped memory areas then, each schedulable object in RTSJ is assumed to have an associated scope stack/cactus that include all the scoped memory areas that this schedulable object has entered and has not ended the execution within it.

3- By using the `executeInArea()` method of this memory area instance object to change the allocation context from the current memory area to be this memory area instance.

According to the RTSJ, the single parent rule must be respected in the both cases; when passing a scoped memory area as an initial memory area, and when the method `enter()` is used to change the memory allocation context. This rule requires that each scoped memory area has to have either zero, or one parent memory area, where zero parent means that the parent memory area is not a scoped memory area, i.e. it is one of the primordial memory areas (the heap or the immortal memory area). On the other hand, the use of the `executeInArea()` method requires that the target scoped memory area has to be *down* within the same scope stack of the schedulable object that executes this method, i.e. the schedulable object has entered it and has not finished execution in it before entering the current memory area.

In the example in Figure 4-3, the scope stack $\mathbf{SM_A}$ is assigned to be the initialization scoped memory area for the schedulable object **X**. So, it comes as the lowest scoped memory area in the scope stack of it when it starts execution. Then, to change the allocation context by entering the scoped memory area $\mathbf{SM_B}$, the schedulable object calls the `enter()` method of this memory area, i.e. SM_B`.enter()`. Then later, if it is running in scoped memory area $\mathbf{SM_C}$, it can change the allocation context to by executing in either the **SM_A** or the **SM_B** using the `executeInArea()` method, e.g. SM_A`.executeInArea()`.

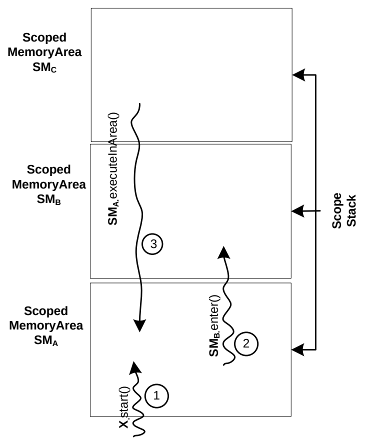


Figure 4-3 Scope Stack and Changing Context Methods

C- RTSJ Memory Assignment Rules

In the memory model of the RTSJ, it is assumed that several memory areas can coexist to offer different allocation contexts for the different schedulable objects created within the RTSJ real-time application. Hence, there is a possibility that an object running within a certain memory area may want to access an object residing in another memory area. However, these memory areas may have different memory management schemes, and objects within them may have different lifetimes. Then a strict assignment rules is placed in the RTSJ on assignments to or from the memory areas in order to prevent the creation of dangling references, and thus maintain the reference safety of Java. These restrictions are summarized and shown in Table 4-1.

As seen in the table, the access to objects allocated in the heap memory area and the immortal memory area is allowed from any other memory area, this is because it is guaranteed that the objects created in these memory areas have a lifetime longer or equal to the calling object. On the other hand, to avoid the dangling references, accessing objects in any scoped memory area is not allowed from objects created in the heap or the immortal memory area. This restriction is because objects allocated in the scoped memory area can have a shorter lifetime than those objects allocated in the heap and the immortal memory. In case of accessing objects in a scoped memory area from other objects that exist in another memory area, the access is allowed with the restriction that the target scoped memory area should be the same as the scoped memory area of the calling object or it has to be within an outer scoped memory area within the same scope stack of it.

Table 4-1 Memory Assignment Rules in RTSJ

	Reference to Heap	Reference to Immortal	Reference to Scoped
Heap	Yes	Yes	No
Immortal	Yes	Yes	No
Scoped	Yes	Yes	Yes, if same, outer, or shared scope
Local Variable	Yes	Yes	Yes, if same, outer, or shared scope

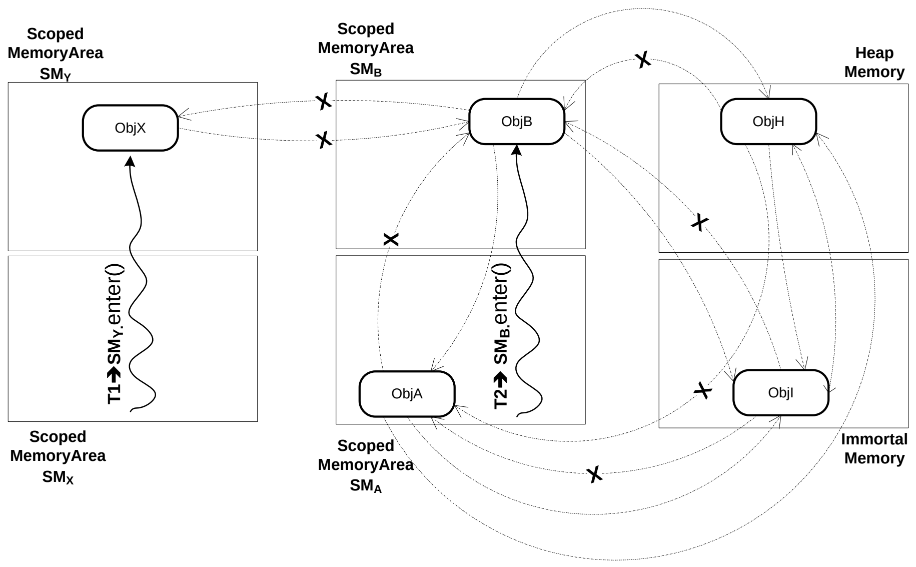


Figure 4-4 Example of memory Assignment Rules

The diagram in Figure 4-4 clarifies these rules. In this diagram **ObjH** and **ObjI** are created in the Heap and immortal memory respectively and they have access to each other. However, these two objects, i.e. **ObjH and ObjI,** cannot access objects **ObjA** and **ObjB** that exist in the scoped memory areas **SM_A**, **SM_B** respectively. In the same example, **ObjX** and **ObjB** cannot access each other as they are allocated in scoped memory areas that are not in the same scope stack. Also, **ObjA** cannot access **ObjB** although it is in a scoped memory area within the same stack scope; however, this memory area is an inner scoped memory area **SM_B** which means that it has a shorter lifetime than **SM_A**. On the other hand, the **ObjA** is accessible form **ObjB** because the scoped memory **SM_A** is in the same scope stack with **SM_B**, and **SM_A** is an outer memory area of **SM_B**.

4.3 Component Based Software Engineering

Development of software systems in general is a complicated task that requires a lot of time and effort along the different stages of the development process. This process is even more complicated for distributed real-time systems due to the predictability requirements and the timing constraints required for these systems. Hence, it is important to provide technologies that support the reusability of the software in order to simplify and accelerate the development of high reliable services, and these technologies are even more important in case of designing distributed real-time systems.

The importance of reusability was known in the early days of software engineering. For example, in (Siddiqui 1996), it is mentioned that in the NATO conference in 1968, a paper was presented with the title "*Mass-Produced Software Components*" in which the author said:

> *"My thesis is that software industry is weakly founded, in part because of the absence of software components."*

Many technologies have been presented to support software reusability. *Subroutines* or *functions* technologies are considered to be the oldest and the lowest level of reusability as they provide a very simple form of reusable code. Over time, the development of newer technologies with higher levels of abstractions helped the emergence of newer reusable entities such as objects which provides a higher level of reusability.

The research of providing software reusability architectures is a research area in the software engineering which is commonly known as component-based software engineering (CBSE), or Component-based development (CBD). According to (Crnkovic 2002), the major goal of CBSE is the provision of support for the development of systems as assemblies of components, the development of components as reusable entities, and the maintenance and upgrading of systems by customizing and replacing their components. This goal requires established methods and processes, not only in relation to the development/maintenance aspects, but also to the entire component and system lifecycle, including organizational, marketing, legal, and other aspects.

As a sub-field of software engineering, there is a strong relation between object oriented programming (OOP) and CBSE. The CBSE uses software engineering principles to apply similar ideas of OOP to the whole process of designing and constructing software systems. For example CBSE focuses on reusing, composing, and adapting existing components, as opposed to just coding a particular style (Siddiqui 1996). Moreover many components models have been built using object technology, e.g. COM, Java Beans, etc., and in these models, the component models adopted object principles of unifications of functions and data encapsulation (Crnkovic 2002).

However, according to (Crnkovic 2002), the difference between OOP and CBSE is mainly in the concept of modeling; where the OOP focuses on modeling real-world interactions and attempting to create "verbs" and "nouns" which can be used in an intuitive way by both the users and the programmers. In contrast, CBSE makes no such assumptions and instead states that developers should construct software by gluing together pre-fabricated components. One more difference lies in the fact that an object has state and it is a unit of instantiation, while a component is a unit of deployment, and it can be either stateless or stateful. A stateful component is a component that can retain information from a call to a next, where the properties of a component hold its state; in contrary, a stateless component has no memory from one call to the next; hence, statless components do not have any public properties.

4.3.1 Component Definition

It is easily recognized from its name that the CBSE is based on using components, so it is important to know exactly what the component is from a software engineering point of view and what its characteristics are. However, it is not easy to give a single general definition for the component that summarizes its characteristics, as the characteristics of the component can depend on many factors including the level of abstraction, the technology used for the design and/or implementation, the specific language or middleware in which the component is used, or even the business applications in which the component is used. Several definitions and different characteristics of components was contributed by many experts were discussed by email and published in (Broy, Deimel et al. 1998). One compact definition of the component was mentioned by Szyperski as follows:

> *"A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties."*

From this definition we see that the internal working of the component is not known to the software engineer, instead he is given only a well-defined external interface from which he must work.

In the same publication (Broy, Deimel et al. 1998), Michael Stal added his own definition as follows:

> *"A component is a binary unit that exports and imports functionality using a standardized interfaces mechanism. The underlying component infrastructure supports composition of components by providing mechanisms for introspection, event handling, persistence, dynamic linking and layout-management. "*

A criticism of this definition was presented in the same publication, because in addition to using some terms those are not clear and need to be explained, it looks to be too dependent on a special technology. So, Michael Stal provided another non-technology biased definition as follows:

> *"A component denotes a self-constrained entity that exports functionality to its environment and may also import functionality from its environment using well-defined and open interfaces. In this context, an interface defines the syntax and semantics of the functionality it compromises (i.e. it defines a contract between the environment and the component). Components may support their integration into the surrounding environment by providing mechanisms such as introspection or configuration functionality".*

4.3.2 Characteristics of the Components

As a software entity, the component has a set of characteristics that are specific to it; these characteristics are dependent on the definition of the component. For example, the component in (Broy, Deimel et al. 1998) was assumed to:

- Represent one or more logical or organizational-related processes or tasks
- Be more coarse grained than single classes; in other words, a component usually consists of several logically coherent classes.
- Be unique from other components because a class can be assigned only once to a component.
- May consist of other components.
- Uses precisely-defined interfaces to communicate with other components.

- Is independent of the release and can be delivered separately.
- Frameworks form the underlying technology of components.
- May be a client and server for other components.

Another view of defining the component characteristics was given in (Wang, Rho et al. 2004), where the authors identified the following three basic characteristic properties of components:

1- **Isolation**; a component should act as a self-contained function unit with well specified interfaces, and it has to be deployable independently as an isolated part, neither the environment nor other components or a third party has access to its construction details.
2- **Compatibility**; the component has to be composable with other components, where other components can access it through a contractually specified interfaces.
3- **Opaqueness;** a component has no externally observable state.

4.3.3 Disadvantages of Components

In order to provide a good design of the component, it is necessary to know the possible disadvantages of using it, so that the design of the component can avoid the scenarios that can lead to these disadvantages. In (Crnkovic 2002) some of the well-known disadvantages of using components were presented and we summarize them as in the following:

1- Components need more time and effort required for development as components are exposed to changes more often than non-reusable parts of software at the beginning of their lives, until they reach a stable state.
2- Components have unclear and ambiguous requirements as it cannot be predicted for all applications in which these components are going to be used in.
3- The design of reusable components enforces adding a lot of complexities that may increase the demands of computing resources; this makes building a simpler and flexible abstract level be more useful in some cases.
4- The component can be applied in different applications with different requirements so, its maintenance support can be high in order to be able to respond to the different requirements of these different applications.
5- Components have separate lifecycles, different than the application in which it will be running and it may have some sensitive characteristics that are not known to the application developer. So, there is a risk that doing changes in the application, e.g.

updating the virtual machine, operating system or other components, may cause system failure.

4.3.4 Basics of Component Based Systems

There are two prerequisites that enable components to be integrated and work together:

1- A *component model* specifies the standards and conventions that components must follow to interact with each other and with the component framework, in order to be that these independent components can be deployed in the composition environment.

2- A *component framework* is the design-time and run-time infrastructure that manages resources for components and supports component interactions. In many respects, component frameworks are like special purpose operating systems that provide the execution and resource management services to the components.

Both of component model and the component framework can be specified at different levels of abstractions, these include:

1- The component model on the level of binary executable and the framework consists of supporting OS services, e.g. COM.

2- The component model and framework are specified on the level of byte code, e.g. JavaBeans, CCM.

3- The component models are specified on the level of a programming language. The framework can contain “glue code“, and possibly a runtime executive, which are bundled with the components before compilation, e.g. koala.

In component based systems, the following terms are defined for the component:

- **The component interface**: the component interface summarizes the properties of the component that are externally visible to the other parts of the system, and which can be used when designing the system. The interface may just list the signatures of the operations, or it can be a *rich interface* that contains additional information about the component’s patterns of interaction with the environment, or extra functional properties such as the execution time, memory use, etc.
- **The component implementation**: this is the executable realization of the component, obeying the rules of the component model, and it conforms to the properties stated in its interfaces.

- **The component contract**. The specification of the functional or extra-functional properties of a certain component is defined as the *contract* of this component. Contracts ensure independently developed components obey certain rules so that the component can interact in predictable ways, and can be deployed into standard build-time and run-time environments.

The contracts were presented in hierarchical representation in (Beugnard, Jézéquel et al. 1999), where contract hierarchy is defined of the following four levels:

1- Level 1: Syntactic interface, or signature, i.e. types, fields, methods, etc.
2- Level 2: Constraints on values of parameters and persistent state variables, e.g. by using pre- and post- conditions and invariants.
3- Level 3: Synchronization between different services and method calls, e.g. expressed as constraints on temporal ordering.
4- Level 4: Extra-functional properties, such as real-time attributes, performance, QoS, e.g. constraints on priority, response time, etc.

Most of the current component models support only level-1 contracts, while some models support others. For the real-time systems, it is important to consider all the four levels, in particular level-3 and level-4, as, by definition, they are directly related to the real-time domain, where level-3 concerns the synchronization and temporal ordering which is timely-based, while level-4 concerns both the timing properties and the quality of services.

Component Based System Development

Component based software engineering uses the software engineering principles and techniques, with a slight difference in the requirements and business goals in the two cases. Hence, there are two different approaches:

1- **Component Development;** the main emphasis in this approach is the reusability, where the components are built for reusability in many applications. So, the component has to be precisely specified, easy to understand, sufficiently general, and easy to adapt, deliver, deploy, and replace.

2- **System Development with Components;** in this approach, the main concern is the identification of the reusable entities and the relations between them. So, the main techniques in this approach include the locating of components, select the most appropriate, adapting them, and verifying them.

In the two approaches mentioned above, different activities can be done independently during the development process. Hence, (Ed Brinksma 2003) distinguishes between the life cycles of these approaches and summarizes their specific activities as following:

Life Cycle of Component-based Systems

The component based systems have the following specific activities:

1- Specify logical and structural system architecture. An early design will create the functional and the logical system architecture, where the architecture specification has to take into account that the system requirements should be compatible with those of available components, so a tradeoff analysis may be needed to adjust the system architecture and to reformulate the requirements to make it possible to use the components.

2- Find and select components that may be used in the system. Available components are collected and investigated to choose among them a set of candidate components, where the selection should be after identifying the requirements of the system.

3- Alternately, create proprietary components to be used in the system. If the available components do not support requirements, so new components have to be developed, which always takes more time than using existing components.

4- Match component requirement with system requirements and verify system properties from component properties. These are basically for components that have rich interfaces that offer extra-functional properties that enable the prediction and verification.

5- Adapt the selected components, so that they suit the requirement specification and the system architecture. Sometimes the component cannot be used directly and it has to be adapted either by using a wrapper code or through a parameterization process.

6- Compose and deploy the components using a framework for components. To obtain a certain function, several components must be composed into a single assembly, these assemblies may result in conflicts between the components, and hence there will be a need for mechanisms for reconfiguring the assemblies.

7- Replace earlier with later versions of the components. New revised components replacing old ones have to have the same interface to be transparent to the system behavior.

Life Cycle of Components

The component development process have many steps of its life cycle similar to those of the system development, the component must be designed, implemented, verified, validated, and delivered. However there are some significant differences as components are built to be reusable, these differences include:

1- There is difficulty in managing requirements, caused by the interplay between component and system requirements.
2- Precise component specifications are more important.
3- Greater efforts are needed to develop reusable units.
4- Verification against component specification must be stringent and documented.
5- In a market for components, property rights and their protection become an issue.

After delivering the component for distribution, the next phase in the life cycle is the deployment of the component into a system, which has to be done without making changes in the rest of the system and this should happen in an automatic way.

4.3.5 Components in Real-time Systems

Component-based system models have proved to be a successful choice for building many software systems as it allows faster developments by reusing tested components that provides the required functionality which minimize the cost and the time of building these software systems. However, according to (Pasetti A and W 1999; Wang, Rho et al. 2004), most existing standards of components such as COM+, EJB, etc. does not address some issues which are essential for real time systems to make it predictable, this makes these technologies unsuitable for use in developing real-time systems.

The set of challenges of developing real-time components was presented in (Isovic, Lindgren et al. 2000) and includes:

1- The real-time component needs to optimize the use of the available target resources, e.g. memory.
2- Many real-time systems have to manage the external load from the external environment; hence, the component must handle the load in a priority driven mechanism that ensures the execution of the tasks of higher priority.
3- Real-time systems have to respond to the events within predictable and specified time limits; hence, components must support predictable event handling mechanisms.
4- The execution of real-time tasks is always bound by timing constraints, e.g. worst case execution time; hence, the design of the task component should provide facilities to manage and handle these constraints.

A wider view of the challenges facing the development real-time component based systems that covers the whole development life cycle of the components was presented in (Ed Brinksma 2003), these challenges are:

- **Component Specification;** it is obvious that the interface specification of the component must cover the four levels of contracts hierarchy; however, there is still no consensus about how components for real-time systems should be specified.
- **Component evaluation and verification;** the trustworthiness of the component in relation to its specification is an important and difficult issue as the components are usually delivered in binary form.
- **Prediction of system properties from component properties;** composing well components with well-defined properties is necessary in producing a system with well-known properties, as the current component models do not provide support for predictable composition.
- **Component models**; component models are the most essential part of the component based systems, for the real-time systems, these models are still in the very early phase of development and the current component models do not support the needs for real-time development.
- **Architecture specification;** the use of components has an impact on the choice of the system architecture, as the available components have to be considered as well as the system requirements when system architecture is chosen.
- **Managing the interplay between achievable system requirements and component specification;** the relations between the system requirements and component requirements are complex, this makes it possible that candidate components usually lack one or more features which is required by the system.
- **Managing changes in component requirements;** the changes of the components and building of new versions may result in conflicts of using multiple versions of the same component within the same system.
- **Update and replacement of components at run-time;** this is an important feature of many real time systems, where the main challenge facing it, is how to combine the optimization of the design-time composition process with this feature.
- **Tool-support;** supporting tools are essential for developing successful component-based systems, various successful tools exist in the non-real-time domain; however, in the real-time domain there is a lack of such tools.
- **Architecture Description languages (ADLs);** these are languages that express the component-based system architectures as compositions of software modules and/or hardware objects, where these description languages concentrate on the description of a system, whose properties, functional and the non-functional, are the composition of properties visible in component interfaces. As the use of component

in real-time systems is not common, then there is a lack of these description languages for real-time systems.

- **Component Repositories;** to allow storing and retrieving components, indexing them, in libraries, and finding the similar components.

From the above, the mission of building a real-time component requires the support of a set of features that overcome the above challenges, but this is not an easy mission as some of the current definitions of the component complicate this mission. For example, some of the component definitions mentioned before characterizes it as a binary reusable unit, this feature is not easy to achieve for real-time components, as the timing behaviour required for the real-time component depends on the target architecture and memory organization on which it has been implemented, so the component may not be portable to other architectures. Moreover, in case of hard real-time systems, the worst case execution time of the task(s) within the component cannot be analyzed and it is not guaranteed that doing the schedulability analysis of the component can give the correct values of it.

Requirements on real-time components

One important issue with the real-time components is that their external interface should be well defined and well specified to enable good use of it. For example, to exchange the information between two components, there are two general models defined in (Isovic, Lindgren et al. 2000):

Buffered (Message Queues); in this model, the two components do not communicate directly, but they use intermediate buffer for communication where the sender component puts the data message in the buffer, then the other components access it. This model, in case of hard real-time systems, requires setting upper bounds on the number of produced/consumed messages through the component interface, to enable guarantee of temporal properties.

Unbuffered (Shared Memory); in this model, the receiving component sends a request first to the client asking for the data, and then the client answers this request and sends the data to a shared memory, which is accessible by the server. This asynchronous model is recommended for hard real-time systems. So, interfaces of hard real-time systems have to be un-buffered.

Another feature of the external interface that is usually needed in many real-time systems is that the access to the external interface has to be limited to a small set of authorized clients, i.e. a security issue, for example the authors of (Pasetti A and W

1999) confirmed that in a satellite control systems, the operation to reconfigure a set of redundant sensors, should only be callable from a failure management system or from the ground control station, so the external interface ideally should have information on which operations can be performed by which clients

As seen from the above two examples, the nature of real-time systems means that it has different requirements from that of non-real-time systems. The general requirements on real-time component based techniques were divided in (Möller, Åkerholm et al. 2003) from industrial perspective of view into:

Technical Requirements

The technical requirements of the real-time component based systems require the real-time component to be:

Analysable; the chosen technique for building the real-time component has to be easy to analyse with respect to its non-functional properties such as timing behaviour and memory consumption, where the component has to be configured at compile time, to ease statically analyses its properties.

Has a standardized modelling language; the design of the component should be based on a standard modelling language like UML.

Open; the component should be in a source code form, i.e. *not* in a *binary* form. This will help the application developers using the component to find functional errors and analyse its behaviour.

Portable; the component should achieve a high level of independency by not using the specific operating system primitives or the processor features directly.

Resource constrained; the distributed real-time systems are always constrained in resources, especially the memory and the CPU, so the component structure should be light weight and its infrastructure has to be minimized.

Development Requirements

From an industrial development point of view, the following features are required in real-time components:

Maintainable; in order to be used for other application or environments other than those in which it has been implemented, the component has to be easy to maintain and change.

Introducible; it is required that the technology of the component to be inexpensive and dependent on or extending existing technologies, so that the companies can migrate to use it easily.

Reusable; the component should be easy to use and the technology used for building and developing it has to support component versioning management, to reduce the risk of reinventing components.

Understandable; the system should be easy to understand in order to simplify evaluation and verification both on the system level and on the component level.

4.3.6 Components Models in Java for Distributed Systems

As we mentioned in the last chapters, Java language supports middleware for distributed systems mainly in Java RMI and CORBA. The component technology found its way in these two middleware solutions, to support components in distributed systems. There are mainly two component models built over these two middleware solutions. These two models are:

- **Enterprise Java Beans (EJB);** this is a server side component model which is based on containers that provides runtime services for managing component activation, concurrency, security, persistency and transactions. The EJB specification defines a component model by standardising the contracts and services offered by the runtime environment and the patterns of interaction between components. The EJB relies heavily on the Java RMI to support dynamic class loading, automatic activation, remote exceptions, and distributed garbage collection, in addition to the support of the transparent distribution of component functionality.
- **CORBA Component Model (CCM);** the goals of CCM are very close to EJB, where CCM is a server side component model that is used to assemble and deploy multilingual components. CCM standardizes and automates the component development cycle by defining a middleware infrastructure and a set of support tools. This architecture uses proven design patterns for handling security, transactions, events and persistency associated with a container infrastructure that enable the access to these services. In its operation, the container is heavily dependent on the CORBA services for distribution component functionality.

4.3.7 Components in Real–time Java

The National Institute of Standards and Technology (NIST) was working in parallel with the authors of RTSJ to outline the required real-time extensions for Java and they state that one of the important goals to be addressed by a real-time Java design is the need of the support of components as black boxes, where they assumed the following to be supported by the real-time Java:

a. Dynamic loading of component code.
b. Component-critical section code should be locally analyzable.
c. The ability to enforce space/time limits.
d. The usage of RTJ-based components should be supported from other languages.

The current RTSJ has not addressed these issues yet and it is still an open area of research that needs a lot of work to be done in order to use the current features of the RTSJ or even change or extend it to satisfy these requirements. Most of the research made in supporting components in RTSJ was targeting the integration of software patterns, and particularly RTSJ-based memory management patterns, to provide simple component models. For example,, a component framework for RTSJ was proposed in (Colmenaresy, Gorappa et al. 2006), in this framework, the components were classified according to the existence of schedulable objects running within them into *passive* and *active* components and they proposed a framework for interaction among these components. Then, in (Hu, Gorappa et al. 2007), an enhancement of this framework was made to provide an XML-based component architectural definition language, and to extend the model to enable composite components definition. The authors assumed that the composite component model is based on the *message passing* design patterns across memory scopes, e.g. the *handoff* pattern mentioned above, shared objects and serialization. In addition, the authors assumed that the communication among parent components and their child components has to be done through a scoped memory manager defined for each component.

In (Plšek, Loiret et al. 2008; Plsek, Merle et al. 2008) another RTSJ component model was provided, this framework is based on the Fractal component model (Bruneton, Coupaye et al. 2006; Coupaye and Stefani 2006). In this model, the classification of components in RTSJ was extended to include *passive*, *active*, *composite*, and *binding* components (cross threads and cross scopes components). Furthermore, the separation of concerns concept was adopted in designing this model, where the design flow of the component model is divided into a business design flow, and a real-time design flow. Moreover, the real time design flow is divided to include a thread management view and a memory management view.

In (Etienne, Cordry et al. 2006), another component model for RTSJ was proposed that also adopted the fractal component model, but their aim was to make the components contract aware, based on the assumptions first presented in (Beugnard, Jézéquel et al. 1999).

Another direction of the development of components in the RTSJ was presented in (T. richardso 2009), in their work, the authors used the RTSJ to study how to provide temporal isolation in the OSGi framework using RTSJ, where they proposed it in two levels: the thread level and the component level. The same authors presented another work in (Thomas Richardson 2010) on adding an admission control protocol to the OSGi framework based on the RTSJ to solve the problem of unbounded dynamism inherited in the OSGi framework. They also presented in (T. Richardson 2010) a model for solving the problems of providing memory management in service oriented architectures. In their approach, they used a technique that allows the calculation of the memory requirements of threads which are used to generate GC parameters, in order to allow the addition of the threads only when it is guaranteed that neither the memory nor the CPU would exhaust.

One more work was presented in (Ruth Tolosa 2003), in this work the authors presented a proposal or a container model based on the RTSJ services; this model was inspired by the EJB model. They provided simple models for invocation, synchronization, resource reservation and memory management based on the available RTSJ services.

Finally, in (Alrahmawy and Wellings 2007) we presented a model that uses movable components in distributed real-time systems. This model was a base of the work that will come next in this thesis.

4.4 Software Patterns

Building software architectures in general, and in particular reusable components and middleware solutions, is a complicated task that not only requires a deep understanding and analysis of the system requirements, but also requires a deep experience of choosing the best design that will provide the most efficient solution that meets these requirements. In general, to build a software architecture, this software has to be divided in smaller units that integrate together to provide the services required by this software. These small units together can be represented in several software forms e.g. modules, classes, components, etc., where each one of these small units needs to provide a particular service or functionality within the system.

Due to the variations of the solutions of the same design problems, several algorithms or design models can be used to build the software units of these solutions; these variations in the solution algorithms have led to presenting some generalized forms of software algorithms as solutions for different kinds of the software units. These solutions algorithms are commonly called patterns and they have been accepted as a mainstream software development technique.

4.4.1 Definition of the Software Pattern

In order to understand what the patterns are and their importance in software developments in general and real-time middleware in particular, we are going to discuss the definition of the patterns, their components, and their types.

Within the patterns community, patterns have been defined in several ways, one very short definition was mentioned in (Lavagno, Martin et al. 2003) to define the design Pattern as:

"A generalized solution to a commonly occurring problem",

This definition is very short and does not give details on the characteristics of the pattern, a more detailed definition of the Patterns was given in (Appleton B. 2000) to define Patterns as follows:

"A pattern is a named nugget of instructive information that captures the essential structure and insight of a successful family of proven solutions to recurring problem that arises within a certain context and system of forces"

Another clear and concise definition of the term pattern was given in (Alexander 1979) as:

"Each pattern is a three-part rule, which express a relation between a certain context, a certain system of forces which occurs repeatedly in that context, and a certain software configuration which allows these forces to resolve themselves"

From all the above definitions, we see that a pattern involves a general description of solution that has been verified to be a recurring solution for a recurring problem with various goals and constraints, where the good pattern will do the following (Appleton B. 2000):

1- It solves a problem not just abstract principles and strategies.
2- It is proven concept, not theories or speculation.
3- The solution is not obvious.
4- It does not just describe modules, but describes deeper system structures and mechanisms.
5- The pattern should provide useful utility for its user.

4.4.2 Classifications of Patterns

The software development process goes through several stages, and in each of these stages the system is divided into smaller units in order to simplify its development, where patterns are applied in each one of these development stages. Hence, one way of classifying the patterns is to classify them according to the stage in which they are applicable in. For example, patterns applicable in analysis stage, are called analysis patterns, whereas patterns applied in the design stage are called design patterns. However, currently in software development, the design patterns are the most popular.

A common way of categorizing the software patterns is according to the software level of abstraction used in the pattern. According to this, the patterns were classified in (Shaw and D 1996) into:

1- **Architectural Patterns;** an architectural pattern is a high level abstract that expresses a fundamental structural organization for software systems by providing a set of predefined subsystems with specific responsibilities, rules, and guidelines for organizing relationships among these subsystems. One example of architectural patterns is the Client-Server model of networked communication in which the system is divided in subsystems such as client object, server object, proxies, etc.

2- **Design Patterns;** a design pattern is a medium scale abstract that provides a scheme for the subsystem of a software system that describes a recurring structure that solves a general design problem within a particular context. The Proxy Pattern which is a class functioning as an interface of another is an example of this kind.

3- **Idioms (Coding Patterns);** an idiom is a low-level abstract specific to a programming language that shows how to implement a particular aspect or relationships of components using the features of an implementation language. The RTSJ’s Portal pattern is an example of an idiom that is defined to enable saving or accessing a shared object saved in a scoped memory area.

Another way of categorizing patterns is classifying them according to the stages of developing them in the software development life cycle as follows (Riehle and Züllighoven. 1996):

1- **Conceptual Patterns;** a conceptual pattern is the pattern whose form is described by means of terms and concepts from an application domain. The Object Pool pattern can be considered as a conceptual pattern if we expressed its concept as a fixed-sized collection of objects that can be retrieved to be used within certain operations of the application and they return back to the pool once they are not required any more, in order to be ready for reuse in other operations.

2- **Design Patterns;** the design pattern is described here by means of software constructs such as objects, classes, inheritance, aggregation, etc. These patterns elaborate upon the conceptual patterns, for example the classes, objects, methods, etc. used for designing the Object Pool pattern described above are considered a design pattern.

3- **Programming Patterns;** a programming pattern is provided in the form described by means of the programming language constructs. This kind is equivalent to the idioms pattern presented before.

One more method of categorizing the software patterns was presented in (Gamma, Helm et al. 1995; Lavagno, Martin et al. 2003) , where software patterns are classified according to their functionality within the software system to be either:

1- **Structural Patterns**; this is the most common type as it describes how classes and objects can be combined to form larger structures. The proxy pattern which is used to implement both the stub and the skeleton of the remote method invocation is an example of a structural pattern, as it acts as a place holder for an object to control references to it, i.e. the stub pattern acts at the client side as is a place holder of the remote server object.

2- **Creational Patterns**; the creational design patterns deal with mechanisms of object creation, by creating objects in a manner suitable to the situation. These patterns are required in problems that can arise when the basic form of object creation could result in design problem or complexity. The Singleton Pattern is a very well-known example of the creation pattern that enable the creation of only a single object instance of a certain class. The Immortal and Heap memory area classes in RTSJ are using the singleton pattern as they are assumed to be singular objects.

3- **Behavioural Patterns**; this type of patterns identifies and realizes the common communication patterns between objects in order to increase flexibility. A well-known example of this pattern is the Observer pattern in which an object registers with a certain entity to observe an event which may be raised by another object. The observer pattern is used for observing incoming client connections in many server models, and in RTSJ it is used by the scheduler to observe the occurrence of the `AsyncEvent(s)` in order to release the AEH(s) registered with these events.

4- **Concurrency Patterns**; Patterns of this kind deal with the multithreaded programming paradigm. The scheduler, pattern is a very common example of this pattern, where it is used to explicitly control when threads may execute single-threaded code, like write operation to a file. This explicit control is done according to a scheduling policy that is encapsulated in the classes and objects of this pattern. For example, in RTSJ, the scheduler is using the pre-emptive fixed priority policy as its own scheduling policy.

4.4.3 Combination of Patterns

As mentioned before, software architectures are commonly built of several software units, and these software patterns have to combine with each other in order to satisfy the system requirements, the combination of patterns can be on several levels from building small software components up-to building full systems. A classification of the levels of combinations of the patterns is defined in (Voelter, Kircher et al. 2004), where the authors classified the following levels of combinations:

1- **Compound Patterns**; these are patterns that are assembled from other smaller patterns, e.g. the Broker pattern used in remote middleware.

2- **Families of Patterns;** these are collections of patterns that solve the same *general* problem, e.g. patterns for networking communication.

3- **Collections or Systems of Patterns**; collections that comprise several patterns from the same domain or problem area, e.g. patterns for supporting server-side non-blocking communication.

4- **Pattern Languages;** pattern languages do not only specify solution to specific problems, but also describe how to create a certain complete domain, e.g. pattern language for *creating* remote communication middleware.

4.4.4 Software Development Using Patterns

In the previous section we discussed what patterns are and how they are classified and combined together. In this section we will show how the patterns can be used. In general, during the design procedure, the patterns can be used in one of three ways as follows (Lavagno, Martin et al. 2003):

1- **Pattern Hatching**; this way of use aims to use an existing pattern from the available pattern models and libraries, by analyzing the design problem in order to characterize its nature and its requirements and whether its scope is architectural or mechanistic, in order to match it with an existing pattern that can provide its requirements and satisfies the set of qualities of service required for it.

2- **Pattern Mining;** it is not necessarily that the design problem to be one of the common design problems that have already an existing pattern to solve it. Hence, a procedure is required to find a new pattern may be needed to deal and solve the problem, this procedure is known as *pattern mining*, and involves abstracting the problem to its essential properties and requirements; then, providing a generic solution to it; finally, studying the consequences of using this pattern in the context in which this new pattern is to be used.

3- **Pattern Instantiating**; the pattern instantiating is the process of applying the pattern to the problem at hand; this involves the specialization of the general patterns roles, modification of application classes to take on properties of the pattern elements, etc.

4.4.5 Patterns in Real-time Systems

One important approach of classifying the design patterns was presented in (Lavagno, Martin et al. 2003). In this approach the patterns are categorized from a real-time perspective, where the design patterns are categorized on the basis of what quality of service (QoS) they seek to optimize into a set of fine grained mechanistic patterns. The mechanistic design pattern is assumed to be a way of organizing some aspect of a design to improve its optimality with respect to one or a small set of qualities of service. Some of these qualities of services that can be optimized by design patterns include:

1- **Performance**; this includes the patterns for optimizing either or both of the worst case execution time and/or the average execution time of subsystems that are implementing these design patterns.

2- **Resource Management;** any software system has a set of resources that are limited and constrained somehow; hence, one of the most important kinds of the optimization required for real-time systems is the correct management and optimization of resource usage, whether this resource can be memory, CPU, hardware, storage, etc. One of the most common examples of hardware resources that have a lot patterns to manage it, is the memory. Memory management patterns include both architectural patterns and behavioural patterns, where the architectural patterns include patterns such as Fixed Allocation patterns, Pooled Allocation pattern, and Garbage Compaction pattern. The behavioural patterns for managing the memory targets mainly the support of the memory sharing in a concurrent environment, such as the Critical Section pattern, to ensure exclusive access to a part of the shared memory and the Priority Inheritance for bounding priority inversion.

3- **Concurrency Patterns**; concurrency is a critical aspect of real-time systems, so the design patterns for managing concurrency are very important in designing real-time systems. The main issues of concurrency patterns are performance and schedulability. In real-time systems, the scheduling patterns include both patterns that:

a. **Represent the scheduling policies**, such as Interrupt pattern, Guarded Call pattern (Douglass 2003), or mechanisms for optimizing schedulability.
b. **Limit blocking and priority inversion**, these patterns overlap with the resource management patterns as the blocking is a result of locking a resource in a concurrent situation. Priority Inheritance and Priority Ceilings are examples of patterns that can be used for limiting the priority inversion.

4- **Throughput;** throughout is a very important quality of service for many real-time systems especially for networked systems. The system throughput for networked system can be specified in one form of the following:

a. **Average Throughput;** the average throughput is average rate of quantity of data that can be transmitted successfully through the system over a certain period of time. For networked systems, average throughput is usually measured in bits/second, and sometimes in packets/second.
b. **Sustained Throughput;** the sustained throughput is the throughput averaged or integrated over a very long time, and it represents the rate at which the data can be sent continuously through the system.
c. **Burst Throughput;** the maximum rate that can be sent through the system at any time**.**

Hence, any pattern that aims to enhance, observe, or control the rate of the transmission of the data in the system is considered a throughput pattern.

5- **Safety and Reliability;** safety and reliability are two important categories for many real-time systems where safety ensures the system freedom from accidents or losses, while reliability is the probability that the system will process the required computation successfully. Patterns for these two qualities of services focus primarily on monitoring the violation of things that are supposed to be always true in the system, *invariants,* and taking appropriate action when such violations are detected. The Redundant Storage pattern is an example of the reliability patterns, where in this pattern the value is stored twice in two different forms, so that it can be validated and corrected if required.

6- **Software Quality, Reusability Patterns;** real-time systems like any software system have some issues that are non-time related qualities of service, such as reusability, correctness, portability, simplicity, scalability, complexity, recurring cost, development effort and cost, maintainability, etc. Hence, all patterns defined for these non-time qualities work for real-time systems as well.

7- **Distributability;** the distribution patterns targets the development of a simple and transparent separation of objects patterns that are located at multiple address spaces and it should provide efficient communication among them, using protocols that maximize the throughput, server response time, reliability, etc. of message transfers among them. Distribution can be classified into:

a. **Asymmetric**; the location of the objects is decided at design time. It is less complex, but suffers from a lack of flexibility.
b. **Symmetric**; the location of the objects is decided at run time, so additional patterns can be built for this type that enable dynamic load balancing of the distributed objects among several processors.

8- **Reactive (Behavioural) Patterns;** patterns of this category define structural elements to control the behaviour of the objects constituting the system. The State pattern (Gamma, Helm et al. 1995) is an example of this category, where it is responsible for controlling and organizing the execution of the different sates that define the object’s state machine.

4.4.6 Software Patterns and the RTSJ

One of the main areas that impact the development of RTSJ applications is the introduction of memory areas into the RTSJ. The constraints and memory access rules required by scoped memory inhibit developers from the direct use of general software design patterns. Hence, there has been a requirement to enhance existing software patterns or even to present new ones that can be integrated with scoped memory areas. Therefore, an active trend of the research in the RTSJ is toward providing new RTSJ compatible patterns, for example, in (Pizlo, Fox et al. 2004) software patterns that predictably execute loops and methods in scoped memory areas were presented, moreover, the *wedge thread* pattern was proposed to keep a certain scoped memory area with shared objects alive even without any schedulable object being active inside it. Furthermore, a *handoff* pattern was presented as a mechanism to enable communication among objects running in different scoped memory areas that have a common outer scoped memory area.

In (Benowitz and Niessner 2003), a survey of software patterns for RTSJ was presented, where an object factory pattern for allocating objects in a specific memory area was proposed. Also, in the same survey, the *memory pools* and *memory blocks* patterns were defined as design patterns for reusing objects especially those allocated in the immortal memory area.

In (Corsaro and Santoro. 2004), the authors presented a memory-scoped version of the *leader-follower* software pattern. In this pattern, a leader-follower selector thread is proposed to be running in a single scoped memory area and can select the leader thread from a pool of threads allocated in the same memory area. Authors in (Corsaro and Santoro. 2004; Corsaro and Santoro. 2005), proposed the *memory tunnels* pattern as an extension to the RTSJ specification; in this pattern, data transfer among objects in different scoped memory areas is done by deep copying objects in a temporary memory tunnel proposed by the authors.

In (Raman, Zhang et al. 2005), the authors presented their experience in using software patterns for developing RTZEN, a real time CORBA Object Request Broker using the RTSJ. The *immortal exception* pattern is one of the patterns they used in their implementation. This pattern provides an exception handling mechanism capable of handling exceptions thrown from objects allocated in scoped memory areas by using reusable exception objects created in a pool in immortal memory.

4.5 Pros and Cons of using Software Design Patterns

Design patterns have a number of practical benefits that make them valuable as tools for software development; however, they also have a number of common problems, which inhibit some developers of using them. In this section, we summarise the pros and cons of adopting and applying design patterns in software development as presented in (Cline 1996).

4.5.1 Practical Benefits of using Design Patterns

Using software design patterns in software development has the many benefits that encourage many developers to use them, these benefits include:

1- **Design patterns coordinate the entire process and community;** they provide standard vocabulary among developers; so, they enable the communication of information between designer, programmer, and maintenance programmer at levels higher than individual classes or functions.

2- **Design patterns can be used reactively;** they can be used as a documentation tool to classify the parts of the design, which makes the design easier to understand, especially for new developers.

3- **Design patterns can be used proactively;** they can be used to build robust designs that consist of well understood design-level parts. This requires the designer to abstract the design problem into parts that can be matched to existing, or new, design patterns.

4- **Design patterns can be used to give the software a hinge;** hinges are required to enable software adaptability to future changes. Design patterns are one of the most useful ways to manage and support hinges, for example, the Abstract Factory pattern, makes the addition of new kinds of derived classes (a hinge) is much easier, even though some parts of the system have to create objects of these derived classes.

5- **Design patterns can turn a tradeoff into a win-win situation;** in many cases, there are tradeoffs, e.g. generality and performance, among some of the qualities of the design. Some design patterns enable the designer to work around these tradeoffs, for example, the Composite pattern provides an efficient mechanism for building recursive structures that are used only when required; this helps to ensure that the design supports generality in a way that affects the performance only when it is used.

6- **Design patterns constrain maintenance programmers;** any change made during the maintenance of the software should not affect the adaptability support of this software. Using documented design patterns helps the maintenance programmers to easily identify and understand the hinges within the code; so that, the maintenance of the system can be planned and implemented without affecting the adaptability of it.

7- **Design patterns let management reward self-directed designers;** To gain the skill and experience, object oriented designers willing to use software patterns in their designs, should expend unpaid overtime hours to self-train themselves how to apply design patterns in simple examples. So, those self-directed designers should be rewarded for their extra effort.

4.5.2 Inhibitors to Patterns Applications

Using design patterns in software development is not always the best methodology for developing software for the following reasons:

1- **Design patterns have been oversold;** design patterns have their importance especially when adaptability is valuable nonfunctional objective. But the benefits of using design patterns are wasted, unless the overall software development process is modified to take design patterns into account. For example, every code review made during the development lifecycle has to respect the constraints of all design pattern used in the design, every maintenance change has to be evaluated to ensure that it does not break any of the constraints of the design pattern.

2- **Some design patterns are unnecessary difficult to learn;** sometimes the chosen design pattern is built over other patterns; which makes it difficult for an average designer to understand it; or sometimes its description/documentation is not clear enough and/or misleading.

3- **Design pattern classifications are not yet useful for practitioners;** for the people who already understand design patterns, the various classifications of design pattern are helpful to organize and search among them; however, some average developers face difficulties of understanding these classifications during the learning stage of using the design patterns. For example, some developers may consider that the criteria used for categorizing the design patterns do not appear to map easily into mental models.

4.6 Summary

In this chapter, as we aim to use the RTSJ in developing our proposed real-time solution, we presented an overview of the new scheduling and memory management models defined in the RTSJ, in order to clarify both the real-time support it adds to the Java language, and the restrictions it adds to the memory model, in order to ensure the required predictability.

Then, as we aim to use the component technology for building our model, we provided an overview of the characteristics and basics of both components and component based systems, and their importance in developing software systems, then we discussed the challenges and problems facing using the component based engineering in the real time systems.

Moreover, we provided an overview of the software patterns as an important software engineering mechanism for developing reusable components and middleware solutions, where we covered their classifications, and the different levels of applying them during system development. Also, we discussed some of the qualities of services that can be optimized using software patterns.

Finally, we discussed the benefits of using the software design patterns for developing RTSJ-based applications, e to overcome the limitations of using the RTSJ’s new memory model.

We conclude from this chapter that component technology is a well-known technology commonly used in building middleware solutions; however, developing real-time components in general and particularly using the RTSJ is facing many challenges, many of these challenges are related to the new memory model of the RTSJ, as the common software patterns used in building components for real-time middleware are not easily implemented using this new memory model. Hence, we need to build a new component model, where this component model can be built using existing and/or new design patterns that have to satisfy the memory rules of the RTSJ, as well as provide predictable execution times and memory usage. These patterns support building the components that can integrate together within middleware models, in order to develop component-based real-time middleware solutions. However, in order to build these components, we first need to define a framework that can be used to build light weight components that can be built over the RTSJ platform and accommodate with the complicated restrictions of using its memory model and scheduling model, So, in the next chapter, we present our proposed framework for developing components in the RTSJ.

Chapter 5

RTSJ-Based Component Framework

The growing complexity of real-time middleware systems has made a need for methodologies to facilitate the design and implementation of such systems. These methodologies should provide high levels of abstractions, to ease the software development, and to enable software reuse. In the previous chapter, we showed that component reuse and modularity features have found great success in supporting fast development of large scale adaptive distributed middleware architectures. This in turn has drawn the attention to using component based software engineering to receive increasing attention for developing distributed real-time systems. Many of the current component frameworks use Java as an implementation language, e.g. EJB. However, these frameworks, as they are Java programs, inherit the unpredictability of the Java language as mentioned in chapter 3. So, they are not applicable in building components for real-time systems.

On the other hand, as discussed in chapter 3, the RTSJ has overcome the unpredictability of the Java language by providing new memory and scheduling models for building real-time applications in Java. To achieve the required predictability in the language, the RTSJ's memory model has some constraints that make it unusual and makes the development of the applications on it a complicated process compared to the standard Java's memory model, which is one of the key advantages of the Java language, due to the automatic memory management provided by the language's garbage collector.

To overcome complexity added by the constraints of the RTSJ's memory model, several design patterns have been proposed in the literature. However, writing these patterns directly within the programs complicates the readability and debugging of the code, and even can cause memory leakage if not properly applied.

Hence, we can see that the above three technologies, RTSJ, design patterns, and the component software engineering technologies, are required to be integrated together to provide frameworks for building components for real-time middleware solutions that have the following characteristics:

1. Hide the difficulties of using the RTSJ memory model.
2. Provide an easy to understand programming model.
3. Facilitate the development of majority of real-time applications.
4. Based on the standard RTSJ features with no additional extensions.
5. Does not add a significant overhead compared to other standard methods.
6. Provide a predictable memory model that does not have any memory leaks.
7. Provide the common services required in real-time middleware architectures, e.g. communication models, where these services have to have predictable execution times.
8. Supports various models of real-time, including static and dynamic architectures.
9. It must provide high degree of configurability and adaptability of its structure to suite different kinds of applications.

Hence, in this chapter, we develop a component framework for building components for real-time middleware applications; we begin the chapter by presenting a design procedure for gradually extracting the components and building the components and objects of the system. Then, we present the design and implementation of our own component framework using a set of existing, modified and new design patterns, which are chosen/created to be compatible with the RTSJ's memory model; at the end of the chapter we present an example of using the framework and its associated sub-components to build a method invoker pattern.

5.1 Components Framework Design Views

As presented in the previous chapter, the elevation of the new RTSJ features, e.g. the new memory model, the different types of schedulable objects, and the new constraints and rules of using them, adds an additional complexity to the design phase of the RTSJ based real-time applications in general, and in particular to the middleware systems that require high levels of abstractions. This added complexity is not just for the provision of the architecture that support the required functionality of these system, but also for embedding the real-time requirements in them using the RTSJ scheduling and memory model. Therefore, there is a need to build a framework for building our proposed real-time middleware, where this framework has to:

1- Use some form of separation of concerns concept to decouple the design phase into several steps, where each step focuses on one type of the system or middleware requirements.
2- Provide new design patterns, or modify the existing ones, to be compatible with the RTSJ scheduling and memory models, in order to provide reliable solutions that can be used in the design of the RTSJ based systems.

Hence, in order to design a component framework model for developing real-time middleware systems using the RTSJ, the framework has to be analysed from different design views that cover the requirements of the environment and the applications in which these components will be used.

In this section, we discuss the different views which we considered when we designed our framework. These design views are extension to the views presented in (Colmenaresy, Gorappa et al. 2006) and the framework presented in (Alrahmawy and Wellings 2007) and they include four basic design views: *Business/Functional View*, *Thread Management View*, *Communication View*, and *Memory Management View*. These design views are explained next.

5.1.1 Business/Functional View

In this design view, the designer focuses only on the functional aspects of the system by dividing the system into a set of functional units; e.g. component or objects, where these units represent the major functional elements of the system including both the passive elements and the active elements. In this model, the active element is an element that contains its own thread(s) of executions, i.e. it uses an active object pattern, whereas a passive element can provide services or require services but it does not have its own threads of executions as it depends on other active elements to provide or use these services. An example of a passive element is a buffer; this object has no threads running in it but it provides reading/writing services that can be used by the active element(s) that wants to access the buffer. Also, in this view, the user tries to specify the required interfaces that should be provided by these elements to use its services, and at the same time, select an initial set of the common design patterns that can be used to build the different services provided by the component from known design patterns, and filter them to the set of the design pattern that can provide real-time behaviour, or find new design.

5.1.2 Thread Management View

In this design view, the functional units created in the business view are filtered, where the passive units are filtered out and only active elements are considered. The active elements can be even classified into single threaded and multi-threaded. Also, as we assume the implementation to be using the RTSJ, the active elements are analysed to determine the execution characteristics of them in order to choose the best schedulable objects that can be used to implement these active elements. For example, the NHRT can be used to implement the active elements with time-critical requirements, whereas the RTT can be used to implement soft real-time tasks. Also, for multi-threading, we have to decide if it is static-multi threading, i.e. the component has a fixed set of threads that runs concurrently during its lifetime, or it is dynamic-multi-threading, i.e. the component has a set of threads that have a lifetimes shorter than the component where they are created dynamically and finishes before the component terminates, e.g. a server component that generates threads to process the clients' requests, these threads are alive only during the processing of the request and each of them terminates once the client request is processed and the result is sent back to the client. This classification may lead to reusing schedulable objects to limit the concurrency within the component.

5.1.3 Communication View

In this design view, the mechanisms required for processing the required communications among the active elements of the system is decided and the components and patterns required for implementing them is added to the components generated in the business view. In general, there are four types of communication mechanisms that can be used within and among the active components/units of the distributed real-time applications:

A- Inner Communication Mechanisms

As we assume that multiple active components can be built by composing a set of active and/or passive components; hence, there is a need to define the mechanisms that can be used for the inner-communication among these internal elements. In other words, these communication mechanisms provide the required composition and integrity of the schedulable objects and their associated passive elements composing a single composite component. The use of the shared memory model is a common example of this type of communication, and it is offered in the RTSJ by either using portals of the scoped memory areas, or through the heap or the immortal memory area. In RTSJ based applications, these mechanisms are mainly dependent on the types of the chosen schedulable objects and it can result in a new set of passive elements added to the design. For example, a wait-free queue, which is a passive element, can be used for inner-communication and binding between a soft-time and a critical-time task that exist within the same component.

B- Inter-Components Communication Mechanisms

These mechanisms are responsible for the communication among the individual components of the application, i.e. these communication mechanisms represent the required bindings among the isolated components that form the application. The techniques used for this type of communication are quite similar for the inner-communication mechanisms, as they both represent communication within the application, i.e. local communication within the same virtual machine. However, they are restricted by the level of communication support offered by the interfaces of the communicating components.

C- Local Communication Mechanisms

These are the mechanisms that offer the active components the ability to communicate across the application boundary with other components in other applications in the same virtual machine.

D- Remote (Network) Communication Mechanisms

These mechanisms are the most important design issues for any distributed real-time system in general and particularly for real-time middleware, as they provide the interface which is used to transfer the data and method calls over a network connection from the active components within an application running on a certain operating system to another application within another node with the same or different operating system. It must be noted that the remote mechanisms used for this type of communication can be used in any other form of communication, i.e. intra, inter-components, or local but with the added cost of time and efficiency.

As presented in chapter 2, in the Java language, remote communication is supported by both the Java RMI, which is Java-dependent, and CORBA, which is Java-independent mechanism. From a real-time point of view, as surveyed in chapter 3, the original Java RMI and the implementations of CORBA for the Java language do not provide the predictability required for use in real-time systems; however, there have been some work to support real-time features in them as presented also in this chapter.

5.1.4 Memory Management View

The memory management view allows the developer to focus on managing the different memory regions of the application. In this view, the different stages of the lifetime of the active and passive units are analysed in addition to the different communication possibilities amongst these units in order to choose the best allocation memory area(s) for each of them, in addition to the memory area(s) that will be used for allocating run-time objects, that are created as a result of executing the active elements. For example, active components that are to exist for the life time of the application are initialized in the immortal memory area, whereas temporary objects that are created during processing a certain task are allocated in a scoped memory that the active component should enter into it while executing this method, i.e. using the encapsulated method pattern, in order to ensure that these temporary objects will be reclaimed once the active component finish executing this encapsulated method. In addition to the memory area assignment, the designer should provide any additional memory management facilities that may be required to manage the lifetime of the scoped memory areas, e.g. the developer may need to use the wedge thread pattern or any similar patterns as will be described later.

5.1.5 Other Design Views

The above design views are not the only views that can be considered when designing the component framework, for example we can consider an additional view, the scheduling view. The aim of the scheduling view is to provide the scheduling mechanism and policies that controls the execution of the active elements of the system. In RTSJ, the scheduling is done using a pre-emptive fixed priority scheduler. Hence, the schedulable objects generated from the previous views can be assigned their priorities and their execution parameters that will be used by the priority scheduler(s) controlling the system. However, for some real-time systems, it might be required to either use other models of scheduling, e.g. EDF scheduling by implementing it over the RTSJ's fixed priority scheduler (Wellings 2004), or even add additional levels of scheduling within the system, e.g. using a distributed scheduling mechanism. In our design of the component framework, we will assume that we are using the priority scheduler provided by the RTSJ, as this is the standard RTSJ model.

The Executable logic view is another view, where logic of the component functionality is addressed and analysed to see how it would be executed and if it will be executed by a single schedulable or it needs multiple schedulable, and how it would be assigned and its execution properties are analysed.

It is seen from the above design views that the design of a component is not a trivial process, and it takes a lot of effort and can be repeated several times to reach the required level of predictability, but we consider that using reusable software components is a good way of simplifying the development of real-time middleware systems, especially by implementing these components using efficient, and reliable design pattern that offers high levels of predictability in both memory usage and execution time. In the next section we present the design of our own component framework.

5.2 The Component Framework

In the component frameworks, the structure of the component and the relations among its parts are developed and the relations and communications mechanisms of the component with other components are defined. This makes using component frameworks simplify the design and development of software systems. In the following sections, we are discussing the main parts of our model and how they can collaborate depending on the different design views presented in the previous section.

5.2.1 The Component Meta Model

The component meta-model represents the summary of the framework as it has to link all the parts of the framework together. In general, the hierarchy of the component should not just define the units forming the internal architecture of the components, but also it has to define the external relations with other components or software elements in the systems, as well as the execution models within the component and how all these elements are integrated together. In addition to these general properties, as this component model targets the real-time domain, in the design of our component model, we are going to consider some important features required for many real-time applications such as the reusability of resources and the avoidance of memory leaks.

Also, as we are building the component using the RTSJ, then the component model has to hide the complexity of the RTSJ's memory model. This requires careful consideration when structuring the component memory model.

We assume that the structure of the component has to be flexible and lightweight to fit the requirements of the tasks that it contains or the functionality it provides. Hence, our design assumes that the structure of the component is not monolithic, in other words, the structure has to be built in such a way that enables the developer to configure its structure to support different hierarchies and services. In the following, we present the basic parts of our component framework.

5.2.2 The Component

The component entity in our model represents and provides a complete unit of service or functionality. In general, the component can have some inner components, and it can itself be part of another component. As said before, the structure of the component needs not to be monolithic; i.e. the inclusion of unwanted structural units within it has to be avoided. So, the structure of the component in our model is based on using individual internal units of structure that provides different services, whether these services are functional services or control services. These internal units do not need to be in all the components, but it can be configured for the components requiring them only, so that these units can be plugged in the component during the creation process as required by the developer. In our model, the component itself is contained within a container that provides facilities and services such as binding, remote loading, etc. to the component.

A- Component Types

The component structure that enables the internal units of the component model to be pluggable only, enables the definition of various types of components according to their structure. Examples of these types include:

1- Structural Sub-Components; these are not independent components, but they are the components that represent a certain structural unit within the component model, in other words any sub-component that does not have an independent functionality and it has to be part of other component(s), e.g. a memory model component, or threading pool component, each of these sub-components can be just a simple object within the container memory area that offers its services to the included components and to other sub-components in the container.

2- Control/Management Sub-Components; these are the sub components that are responsible for managing the structural sub-components constituting the component hierarchy, e.g. a memory lifetime controller that manages the lifetime of the inner scoped memory areas, as the structural sub-components, these can be just objects in the container.

3- Active Components; these represent components that have inner executing tasks, in other words they have their own executing thread(s) and they can work either independently or cooperatively with other components. This type of component can be even divided into two main categories:

a. **Static-Structure Components**; these components have a static structure that does not change along the life-time of the component, e.g. a component that encapsulates a timer or a client in a middleware that makes a request to the server and waits for results.

b. **Dynamic-Structure Components;** the structure of these components can be changed over the life time of the component, i.e. other components are attached to it or other tasks components are generated within it, e.g. a server component that handles connection from clients, where it can create inner tasks as inner components to handle the execution requests of the clients.

4- Passive Components; these represent components that have no internal thread of execution, and hence they cannot work by their own independently of other components. In other words, these are (*hospitable*) components that enable the execution units of other *(guest)* components to execute within them, e.g. a buffer component.

5- Logic Components; these are components that represent the execution logic of a certain task. In other words, they have neither inner memory nor threads of execution, so that they have to be attached to a certain active component to run within it. The use of the logic component enables the flexibility of the execution of the logic by not making it tied to a certain executing component, which gives the flexibility for the logic to be executed by more than one executor, e.g. the migration of the code in mobility application.

In this chapter, we will show some of these components and how they integrate within the framework.

B- Component Design

In our framework model, we assume that the component entity is represented by the class `ComponentsCls`, so that any component has to be created from a class extending the `ComponentCls`. The `ComponentCls` class implements the interface `IComponent`, shown in Figure 5-1. The interface `IComponent` defines a set of functions which can be classified into four different groups:

Group 1: This group defines the management operations of the component, and it includes four different operations:

`start()`. To start the execution of the inner schedulable tasks of the component; if they exist.

`init()`. To do all the initialisation operations required for the component.

```
package RTCOM;
public interface IComponent
{

//Group 1
       String getComName();//retrieves the name of the component
       void start();//start the driver thread of the component
       void init();//initialize the component
       void terminate();//terminate this component
       void setContainer(IContainer container);//assign a container
       IContainer getContainer();//get the container of the component
       MemoryArea getCMA();//get the common memory area of the component
       MemoryArea getContainerMA();//get the container memory area
       Queue getSMAQueue()//get the Queue of SMAs
//Group 2
       void initMemoryModel(IMemoryModel memModel);//initialize the memory model
       void initMemoryModel(Class memCls, long initSize, long MaxSize,
                 int LIMIT,Class SMAClass, long[] initialSizes,long[] maxSizes,
                 Class memCtrlCls );//initialize the memory model
       void initMemoryModel(LTMemory theCMA, int LIMIT,Class SMAClass,
                 long[]initialSizes, long[] maxSizes,
                 Class memCtrlCls );//initialize the memory model
       IMemoryModel getMemModel();//retrieve the memory model

//Group 3
       Schedulable getTask(String TaskName);//retrieves a certain task by name
       ReusableRunnableStack addSMATask(String taskName, Class SchedulableExecutor,
                    SchedulingParameters schedulingP, ReleaseParameters releaseP,
                    long scopeSize, long immortalSize , Class mareaType,
                    ProcessingGroupParameters group,
                    Class StackLogicCls);//adds SMA task
       ReusableRunnableStack addSMATask(String taskName, Class SchedulableExecutor,
                    SchedulingParameters schedulingP, ReleaseParameters releaseP,
                    ScopedMemory memArea, ProcessingGroupParameters group,
                    IStackLogic logic );//adds SMA task

//Group 4
       IMemoryModelControler getMemoryController();//get the memory controller
       void setMemoryController(IMemoryController
                      memController);//assigns memory controller
       void setMemoryController(final Class
              memControllerCls);//assign memory controller

       IObjectAllocator getObjectAllocator()//get the reusable object allocator
       void setObjectAllocator(Class memoryControlerCls)//assign object allocator
       void setObjectAllocator(Object memoryControlerObj)//assigns object allocator

       IHPool getHandlerPool(String poolName);//Get the pool of handlers
       void addHandlerPool(String poolName, IHPool pool);//add a handler to the pool
       void addHandlerPool(String poolName, Class poolCls);//add handler to the pool
}
```

Figure 5-1 the IComponent Interface

`terminate()`**.** To terminate all the schedulable objects running in the component, and then terminate the component itself.

`void setContainer(IContainer container)`**:** to assign the container that holds this component.

`IContainer getContainer()`**:** to get a reference to the container that holds this component.

`MemoryArea getCMA()`**:** to get a reference to the common memory area, CMA, of this component.

`MemoryArea getContainerMA()`**:**to get a reference to the container memory area, ContMA, of this component.

`Queue getSMAQueue()`**:** get the Queue that holds all the forked/pinned scoped memory areas within the CMA of this component.

Group 2: This group includes all the operations responsible of the memory model assigned to this component including operations for initializing it or retrieving a reference to it.

Group 3: This group includes all the operations responsible of the adding and getting attaching schedulable objects (reusable or not) to/from the component.

Group 4: This group includes all the operations responsible for the adding or accessing the optional sub-components to/from the component or its container.

An example of a basic implementation of the `ComponentCls` class is presented in A.1.

5.2.3 The Container

The container is a common structural unit used in many component frameworks; the main function of the container is to manage the lifecycle of the component i.e. it provides the control interfaces that enable starting, initializing and terminating the component. In addition to that, the container offers a set of services to the components running within it, including communication, resources reservations, etc. The container itself has its own memory area in which all the main memory areas of the inner components are allocated. In our model, we consider the container is acting as a host of the component that isolates it from other containers, components and/or objects in the program. Also, we consider that the container is assigned a thread that is responsible for managing the component and its inner component lifecycles, i.e. *Manager/Driver Thread*.

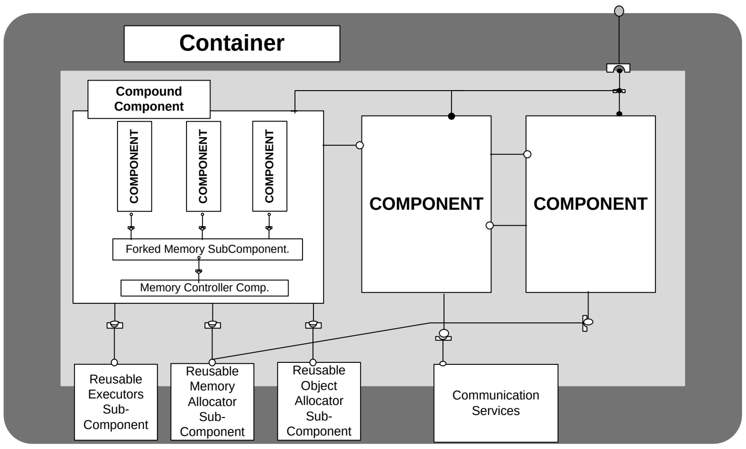

Container
Compound Component
COMPONENT
COMPONENT
COMPONENT
Forked Memory SubComponent.
Memory Controller Comp.
COMPONENT
COMPONENT
Reusable Executors Sub-Component
Reusable Memory Allocator Sub-Component
Reusable Object Allocator Sub-Component
Communication Services

Figure 5-2 The Container and its inner components

We assume that the container can have multiple components inside it, see Figure 5-2, either as a single composite component or as a set of components or attached (communicating); where this container can provide one or more of wrapper interfaces that enable the external access to the services provided by its inner components. Moreover, the container in our model works as a provider for the optional reusable resources and communication services required for the component(s) running within it that require them. The container provides these reusable resources; e.g. object pools, ... etc. in the form of structural sub-components composed within it as will be explained later in this chapter.

In our model the Container is implemented using a class that extends the abstract class `ContainerCls` which implements the `IContainer` interface, shown in Figure 5-3.

```
package RTCOM.
public interface Icontainer
{
       ScopedMemory getMemArea();//Get the memory area of the container
       HashMap getComponents();//Get the components in the container
       void addComponent(IComponent com);//add a component to the container
       IComponent getComponent(String ComName);//Get a component from the container
       void BuildComponents();//Build the inner components
       void InitializeComponents();//Initialize the inner components
       void start();//start the execution of components in this container
       void terminate();terminate the execution of components in this container
       …..
       ….

}
```

Figure 5-3 The IContainer interface

`void InitializeComponents()`:To do the initialization phase of all the components running within this container, by calling the `initialize()` function of each component.

`void start()`: To start the execution of all the components running within this container, by calling the `start()` function of each component.

`void terminate()`: To terminate all the components running within this container, by calling the `terminate()` of each component.

In addition to the above functions, the class that extends the `ContainerCls` in our model can provide the additional interface, e.g. `IContainerServices` that gets references to the set of optional sub-components that are created, within the Container, e.g. memory pools, etc. this interface should define a set of functions like: `getMemoryPool(), getObjectAllocator(), etc.` An example implementation of a class that represents our model of the Container is presented in A.2.

5.2.4 Interfaces

Interfaces play a major role in component models; they represent the entry to the component services and structures. In our framework, each component can have one or more interfaces which can be categorized into the following:

- **Information interface(s)**; one or more interfaces can define certain information of the component like its name, id, etc. In our view, these can also include the information that defines real-time and predictability properties of some inner tasks of it, like its worst case execution time, maximum memory, etc.
- **Control Interfaces**; these are the interfaces that are responsible for managing the inner configuration and operation of the component, i.e. its start, termination, size of its memory, or type of its threading model, etc.
- **Functional/Business Interfaces**; these are defining the different services offered by the component, i.e. these are the ones that are defining the component job, so we assume that each component has at least one interface that defines its functions.
- **Communication interfaces**; these interfaces are defining how the component is linked to other units in the program, including other components in the same container, or other components in another component i.e. binding interfaces, or even with other components in other applications on the same machine or other machines, e.g. remote interfaces.
- **Scheduling Interfaces**; these are the interfaces that are managing the scheduling of the component inner schedulable objects, or even the schedulability of the component itself in case of using a multi-level scheduling mechanism that enable scheduling the components, i.e. the mobility scheduling of components over a distributed system (see (Alrahmawy and Wellings 2007)).

In our model, it is not necessary that each interface is to be of a single specific category of the categories mentioned above, but it is possible that the same interface can be a combination of more than one category.

5.2.5 Memory Model

As one of the main aims of designing our component framework is to abstract and hide the complexity of the RTSJ that results from its unusual memory model and its associated rules and constraints, then, in our design we consider that the cornerstone in our component model is its memory model. This memory model should be built in such a way that it optimizes the use of the memory structure and the associated memory management and communication design patterns. Then, from this memory model, we derive the other elements that can integrate with this memory model to build the component.

So, here we present the memory model for our proposed component model, where we will first define a structure model for the component itself, and then we will discuss how we can provide rules and mechanisms over this model to present a full memory management model of the proposed system.

A- Component Memory Structure Model

In our system, from a memory structure point of view, there are two types of components:

- **Memory-Owner Components**; these are the components that have their own memory structure, which they use during the lifetime of the component. Examples of these components include server components that provide services to multiple clients.
- **Memory-Guest Components;** these are the components that do not have their own memory structure, but they are using the memory structure of other memory-owner components, e.g. the control sub-components that control the life time of other components.

In our system we assume that, regardless of being a structural or functional component, there are two general types of components active components and passive components. The passive ones do not have any schedulable objects running within them, but they have a memory structure and they allow schedulable objects from other components to access their memory structure, e.g. as in the case of defining a buffer component, or they can be guest components that do not have their own memory structure but they are using the memory structure of some other components, e.g. the reusable logic components that are assigned dynamically to other components that execute the logic they are defining.

On the other hand, the active components can have one or more schedulable objects depending on the functionality of the component. For a component that encapsulates a single task, the component can have a single schedulable object that runs on a simple memory structure that can consist only of a single memory area, while for the component that encapsulates multiple concurrent tasks that integrate together to provide the functionality of the component, the component can have multiple schedulable objects running inside it with different memory areas assigned for these schedulable objects. Also, the components that have multiple concurrent tasks can have either a static structure or dynamic structure depending on the functionality of the component, for example a component acting as a server for multiple clients can have a dynamic structure by creating dynamically a schedulable object to handle any new received request, or it can have a static structure by queuing the requests and processing them in a certain order using the same schedulable object handler(s).

So the memory model has a general structure that supports one or more tasks that can be running concurrently and collaboratively within a single component to define its functionality. At the same time, the memory model that represents this component model has to respect the RTSJ memory model and its assignment rules. In order to provide this component memory model, we proposed the following, see Figure 5-4:

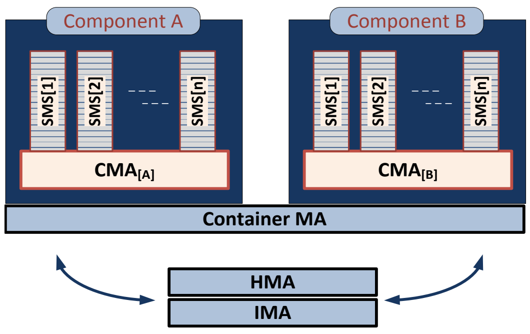


Figure 5-4 Basic Component Memory Model

Every component has at least a single component memory area (**CMA**), where in case of the non-real-time components, i.e. components that have only normal Java threads; it can be the Heap or the Immortal memory area. But for the case of the real-time component, i.e. a component that has one or more schedulable objects, its CMA has to be either immortal or scoped memory area. In the next steps, we will restrict ourselves to the real-time component case, as this is our main concern, as there is no restriction on using the heap memory with the normal Java threads.

During the component creation procedure, the component CMA is created by the container's *Manager Thread* as a memory area instance from within the component's container, where we are assuming that the container memory area (**ContMA**) will be a parent memory for all the CMAs of its internal components. This assumption is important to ease the communication among the components within the same container as will be discussed later. An exemption of this assumption is for some of the sub-components that have to exist in the container memory area, to provide services to the other components; these sub-components have their CMA as the ContMA itself.

For the case of passive components, we assume that the component must be assigned at least one memory area that defines the context in which the objects of the component are to be allocated, and schedulable objects from within other components are allowed to *enter* this memory area, subject to the restrictions of the RTSJ memory access rules, and they can build internal scope stacks inside it.

All schedulable objects within a single real-time component are initiated so that they are sharing both the container's ContMA and the component's CMA as their deep most parent memory areas, This assumption ensures that:

1- All the schedulable objects in a single component are sharing the same CMA, so they can use it for communication among them.
2- All components in a single container are sharing the same ContMA, so that they can use it for communication, if required.
3- As in our model, we assume that the container can provide pools of recyclable schedulable objects, then, initiating all the schedulable objects from within the container supports that all the schedulable objects within the same container can be reused within any of the components sharing this container where all idle or non-used recyclable objects are created and kept waiting in the container until they are assigned to a certain task in a certain component in this container.
4- During component execution, each schedulable object can have its own scoped memory stack (SMS) that enables it to execute in different memory areas other than the CMA and the ContMA, to avoid the creation of the redundant objects in the CMA and the ContMA. This allows each schedulable object to use its own scoped memory areas for temporary object allocation according to its demands in order to optimize the memory allocation, while allocating the component's common objects that are valid along the lifetime of the component in the CMA memory area.
5- We assume that the scoped memory stacks of the active components (one or more nested scoped memory) are exclusive, i.e. a single scoped memory stack cannot be shared among several components, while the scoped stacks of the passive components are sharable by multiple active components to enable communication among the components, where these schedulable objects have to respect the single parent rule when they access these shared passive components.
6- According to the RTSJ memory model, the immortal and heap memories are accessible from any schedulable object of the component that is running in any memory area within the component, this allows that the class data and static objects are always allocated in the immortal memory area. Also, the heap and immortal

memory can be used as a shared memory area for communication with other component executing in the same JVM, except with the components that have no-heap objects which have to use the immortal memory or cross scoped methods for communication.

7- In the case of the simple component that has a simple function that is represented by single task, the component memory model presented earlier in this chapter is simplified to the model shown in Figure 5-5, which has a single scope stack with the CMA, ContMA as the most outer memory area of it.

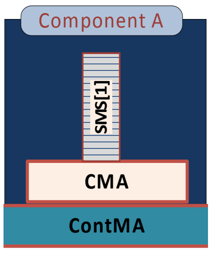


Figure 5-5 Simple Component Memory Model

5.2.6 Component Composition and Communication

In our component model, the component itself can be composed of a set of integrated components, i.e. structural components such as a memory pool component that integrates with memory structure components and other components to form the required *functional* component. In our model, the functional components themselves can be integrated together to either provide us with the ability of the construction of composite components and/or to build a system of communicating components running together in the same JVM. This is important for building real-time middleware solutions, as the middleware solutions are usually built of several software units that integrate and collaborate together to provide the required middleware structure. In the following section we will discuss the different models of composing and integrating the components in our framework.

A- Functional Composite Component

The functional composite component is assumed to be a component that has multiple internal components running within it. In our model, we assume that each composite component can be build inside a single container that encapsulates all the required components. According to this assumption, we can derive two different models of the functional composite components:

1- **Shared Memory Model**; in this model, see Figure 5-6, the internal components, C_1, C_2... C_n have all the same component memory area CMA_C; hence, this memory area can be seen as the common memory area of a single functional composite *component* that embeds these internal components.

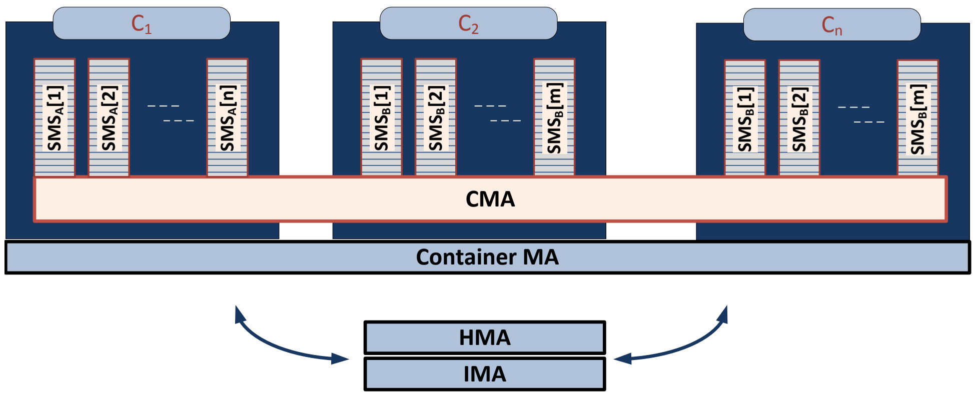


Figure 5-6 shared Memory Model

If we consider that each inner component of this model has only a single scoped memory area stack (SMS), then the model will be reduced to a model equivalent to our assumed basic component model as seen in Figure 5-7, where each inner component of the composite component will simulate a single SMS model of the basic component.

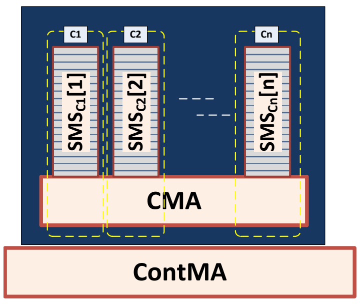


Figure 5-7 Simple Composite Component Model.

2- **Isolated Memory Model;** in this model, see Figure 5-8, each component of the internal components, C_1, C_2... C_n has its own component memory area CMA_i. Hence, they do not share a common memory other than the container's memory area ContMA. Hence, the ContMA memory area can be seen as the common memory area of the functional composite *component* that embeds these internal components.

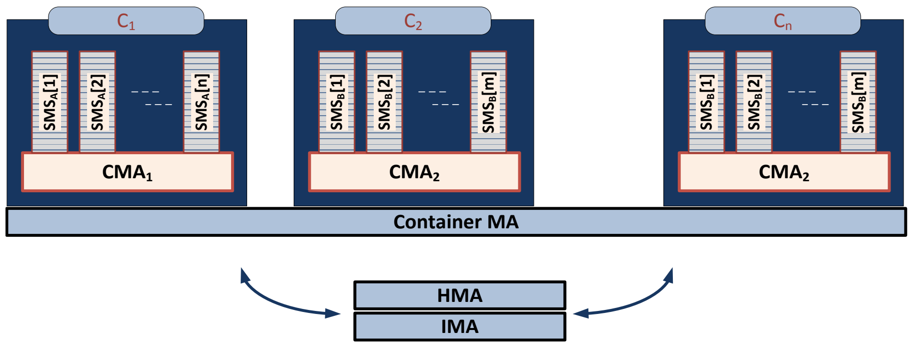


Figure 5-8 Isolated Memory Model

B- Components Binding

In our framework model, we assume that multiple components can coexist in a single container in order to build a composite component. This requires these components to be linked to each other in order to support the communication among them. We proposed a simple way of linking each two components through the interface injection pattern (Etienne, Cordry et al. 2006), in this pattern as shown in Figure 5-9, the class of a component that has dependency on other component has to implement the `IBindController` interface that has two methods:

`bindInterface():` it takes a name of an interface and a reference to the class that implements it. The function checks if the given interface name is one of the required services for it, if so, it assigns the given component reference to the corresponding private references in the class.

`unbindInterface():` this function just removes the reference of the component that was linked to a certain interface within the class of the component.

C- Inner/Inter Components Communication

Inner and outer communication mechanisms represent the communication among the components residing in the same virtual machine. In this section we present the communication models between each two individual components in the same JVM as defined in our framework. In our framework we can define the following models of communication in the same JVM:

Inner Communication through the Shared CMA Model

In the shared memory model of the composite component defined above, two or more individual components are initialized with the same common memory area, so that any communication can be done through objects allocated within this shared memory area. As seen in Figure 5-10, the two composite components ***A, B*** are created with the same common memory area *CMA*. Hence, all the sub components *C1, C2* ...

Cn from component ***A*** can communicate easily with any of the sub-components *Ca*, *Cb* … *Cz* of component *B*, through this memory area, by allocating shared objects in it to be accessible from both components through the CMA's portal. As the life time of the CMA is the life time of the both components, then objects created in it has to stay there for the life time of the two components, or they have to be allocated from a reusable pool of objects saved in the CMA.

```
public class myComponent extends Component implements IBindController
{
     private IReaderWriter irw;//ref to an external Component
     private ICommunicator icom;//ref to another external comp
     //The following method binds a certain component/object
     //to a a corresponding reference in the class
     public void bindInterface(String itfName, Object component)
     {
          If(itfName.compareTo("ReaderWriter")==0)
              irw=(IReaderWriter) component;//bind the first component
          elseif(itfName.compareTo("Communicator")==0)
              icom=(Icommunicator) component;//bind the second component
     }
     //The following method unbinds a reference of the component
     //given its name itfName
     public void unBindInterface(String itfName)
     {
          If(itfName.compareTo("ReaderWriter")==0)
              Irw=null;//free the ref
          elseif(itfName.compareTo("Communicator")==0)
              icom=null;//free the ref
     }
}
```

Figure 5-9 A class implementing the `IBindController` interface

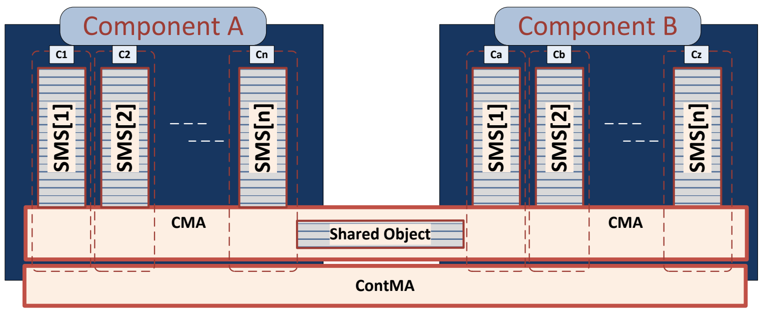


Figure 5-10 Communication through a the CMA shared memory

Inner Communication through the Shared ContMA Model

This model is exactly the same as the previous one, except that the communication is done through the container's ContMA itself, not the CMA, see Figure 5-11. This makes this model applicable to the both models of composition defined above, i.e. the shared memory and the isolated memory models. However, in the case of the shared CMA model, the use of the shared CMA for communication offers higher level of isolation and encapsulation for the communication process as it isolates it from the other components that exist in the same component but do not share the CMA with the communicating components.

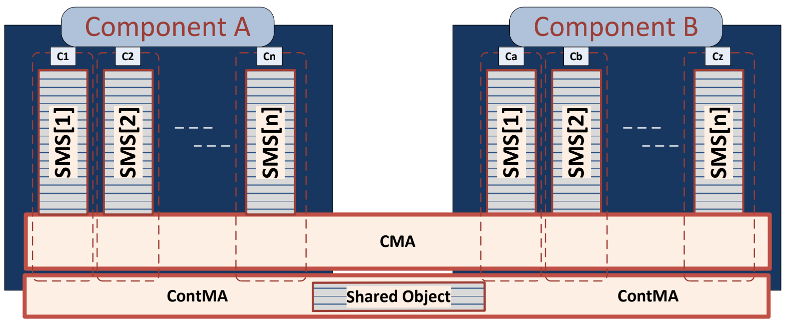


Figure 5-11 Communication through a the ContMA shared memory

Inter Communication through an Attached Scoped Memory Model

The shared memory model is relatively easy to use, but the creation of the shared object in the CMA or the ContMA enforces the need for techniques for reusing these shared objects if they are frequently and dynamically created. In our model, we can overcome this limitation, we can use the model shown in Figure 5-12, in this model it is allowed for any schedulable object either to use an existing passive component with a SMS attached to its CMA and or ContMA, or to create an external scope stack **$SMS_{[\Psi]}$** which can be seen as a virtual passive memory buffer component C_Ψ that enable the creation of temporary objects that can be accessed by schedulable objects of both components ***A, B*** using the handoff pattern (Pizlo, Fox et al. 2004). This virtual memory component can be used to save the required object(s) used for communication, where they can be created by any of the schedulable objects that share its parent memory area, i.e. CMA and/or ContMA. However, the use of this attached memory requires a special handling of the framework as it requires at least one thread to be running in it to be able to keep the shared objects inside it alive. This can be managed by using one of the models of the memory lifetime management control presented later in this chapter.

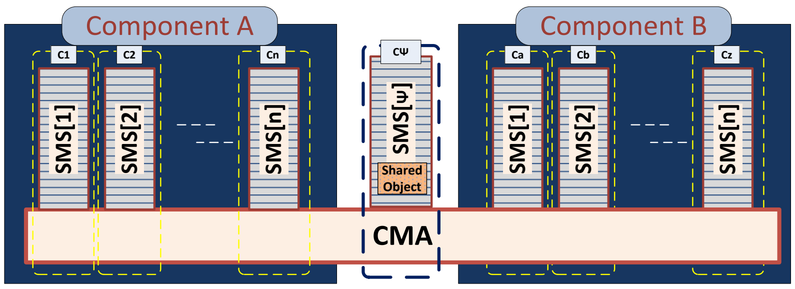


Figure 5-12 Communication through an attached scoped memory area

Inter Communication using Attached CMAs.

In this model, any component can access the memory of the other component by attaching the other component's scope stack as an inner scope stack. For example, in Figure 5-13 and Figure 5-14 , the *component **A*** is attached to *component **B*** by one of two ways:

1- ***Component B*** is created dynamically by one of the scheduler objects of ***Component A*** when this scheduler object is running in its component memory area $CMA_{[A]}$ and assigned $CMA_{[A]}$ as its container's memory area, see Figure 5-13. .

2- ***Component B*** is entered from within one of the scheduler objects of ***Component A.*** when this scheduler object is running in its component memory area $CMA_{[A]}$, see Figure 5-14.

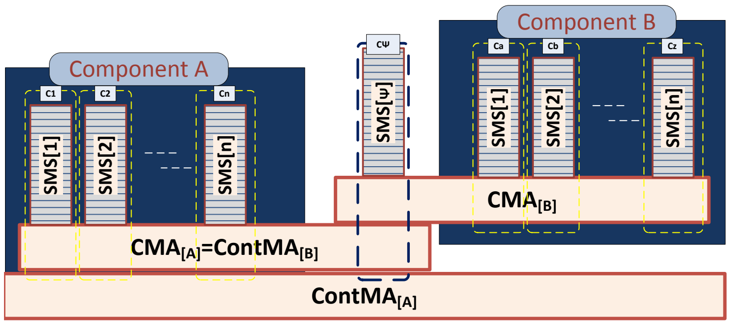


Figure 5-13 Component Attaching – case (a)

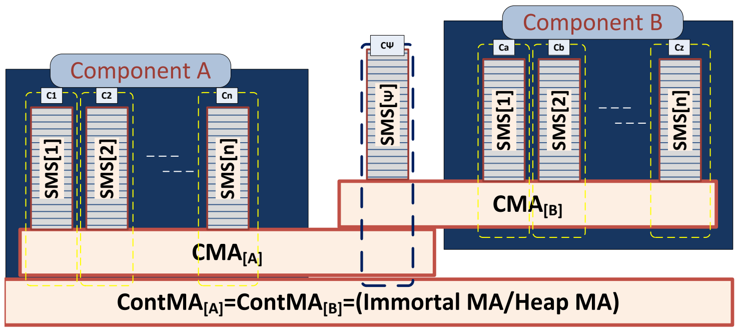


Figure 5-14 Component Attaching – case (b)

In these two cases, the $CMA_{[A]}$ will be a common memory area of both components, and any communication between the two components can be done through it in the same way as stated before. However, in the second case, i.e. using enter, the RTSJ single parent rule must be considered, i.e. it requires that the $CMA_{[B]}$ to have no other parent scope, other than $CMA_{[A]}$ to make the attachment possible. This can be only valid if the two components have the similar container which has the immortal memory area or the heap memory area as its memory area.

In this model, like the previous model, a virtual passive component CΨ can be created dynamically to enable the communication; however, in this case, this a virtual passive component will have a scope stack with at least three memory nested memory areas, the ContMA$_{[A]}$ as the most outer scoped memory area, the CMA$_{[A]}$ itself as a second outer most, then the CMA$_{[B]}$ as a third most outer scoped memory area, objects created during the communication in all of these three scoped memory areas will be alive even after finishing the communication as long as the components are alive. So, their inner scoped memory areas are used for allocating temporary objects during the attachment.

We must note in both cases above, i.e. using shared memory model, or attachment model that the communication among the schedulable threads can be done through a virtual passive component that can be accessed by both of the two communicating components. However, this passive component may have a stack scope that, according to the RTSJ rules need at least one schedulable object running within it to ensure that it will not be de-allocated. So, techniques to keep this virtual passive component need to be used, e.g. wedge thread. We discuss some of these techniques later in this chapter.

D- Local/Remote Components Communication

Local and remote communication mechanisms represent the communication among the components residing in different Java virtual machines. In this section we present the communication models between each two individual components in different JVMs as defined in our framework.

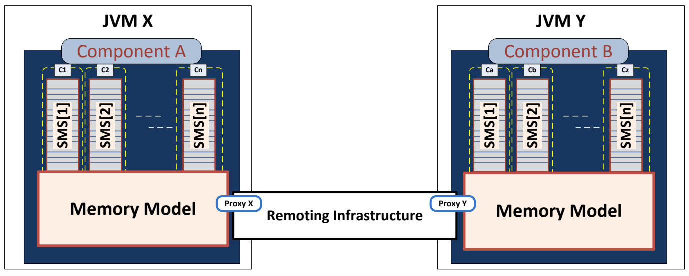


Figure 5-15 Remote communication between components

As presented in chapter 2, there are several paradigms that allow remote communication among objects/components on different JVMs. Most of these paradigms require a remote communications infrastructure with proxies for doing the communication operations on behalf of the objects/components themselves as will be declared in details on the next two chapters. We can classify these levels of communication into two levels:

1- **Low Level of Communication;** this level represents the communication between the proxies themselves. In this level, the two proxies handle the communication in a form of exchanged packets over the network, where the sender's proxy receives the request from the sending component and packs the request and the related parameters in a byte form and sends it to the other receiving's proxy that receive the packet, decode it, and forwards it to the target component to run the request, then it waits for the result to return it back to the proxy at the client side.

In this model, we suppose that the proxies should have lifetime at least equal to the components using them; hence, the choice of the memory region in which these proxies will be allocated can be:

- In the component memory area of the component, CMA, this is in case if the proxy is servicing this component only.
- In the immortal memory area (in the real-time case) or heap memory area (in the non-real-time case), if this proxy is shared among several components in different containers in the application, i.e. it offers its services along the lifetime of the program.
- In the container memory area, ContMA, this is in the case that this proxy will be used by one or more components within this container.

In general, the second and third choices are more efficient as in addition to using a single proxy for multiple components; it can be built as a service that runs independent of the component itself.

However, to support concurrent use of this low level communication service, this requires:

- The proxies have to be able to multiplex/demultiplex the different requests;
- The low level communication infrastructure needs to be implemented as separate component/sub-component with minimum binding with the components using it.
- As it is responsible for the real-time communication in the system, the proxies should offer high predictability and efficient communication mechanisms, and at the same time should be predictable in their memory consumption.

In chapter 6, we will provide our model for building a communication component that support a low level-communication in our proposed component model.

2- **High Level of Communication;** this level is built above the low-level mentioned above, and it provides a communication infrastructure between the schedulable objects of the communicating components. In this level, the communicating components exchange Java objects in a form of sent parameters and/or return results. In our model, this level requires that:

- It has to be subject to the restrictions of the RTSJ memory model.
- The definition of paradigms that enable the easy use of the underlying proxies.
- It has to support different communication schemes to be useful for several kinds of applications, e.g. synchronous calls, asynchronous calls, etc.

In chapter 7, we will go into the details of designing a component that support the high level of remote/local communication.

5.3 Memory Model Implementation

In the previous sections we discussed the basics of our proposed component framework, where we showed that the component itself can be built from smaller structural components, i.e. subcomponents like the memory model, the object allocators, etc. In the following, we are going to provide the set of the structural components that are used to build the memory model of our proposed component model. We will focus on defining the software pattern used for building the basic memory model for the components, and then we will provide some software patterns that integrate with it to manage the lifetime of this memory component to overcome the life management problems of RTSJ's scoped memory areas and to provide memory management services within the model.

Beside the usual requirements of designing components in general (i.e. configurability, integrity with other components, etc.), designing a model for components based on RTSJ must also take into consideration both its memory and threading models. This involves the patterns and techniques used to construct the internal elements of the components and how they can be integrated together to satisfy the general constraints defined in the RTSJ. Hence, from our viewpoint, the structure of any memory model based on the RTSJ should satisfy a set of requirements that include:

1- it must be constrained to conform to the RTSJ memory access rules,
2- it must provide an efficient use of memory resources with minimum overhead,
3- it has to avoid the redundant use of resources,
4- it has to provide a coherent model that is easy to build and to use,

5- It has to support a high degree of isolation from the other components to ease operations such as maintenance and replacement.

In the following sections we will present our proposed new memory model, ***the Forked Memory Model***, we develop this model as a simple sub-component that can be hosted within the component model defined in the previous sections. Then we will present a set of sub-components that integrates with the memory model subcomponent to manage the lifetime of scoped memory areas and reusable shared objects using some new software patterns. After that, we will present another set of sub-components that support the using reusable schedulable objects and reusable logic patterns within this memory model.

5.3.1 A Memory Sub-Component Model

In order to develop our proposed *Forked Memory Model*, the following assumptions have been considered:

1- The component can consist of one or more tasks co-operating together to perform a single integrated service offered by this component, where the task can be either a single thread that represent an operation to be executed by the component or an event handler that handles a certain event occurring within the component. In other words, each task can be executed mainly by a single schedulable object.

2- The task set within the component model can either be static, i.e. they are created at the start and initialization of the component as a fixed set and no other tasks are added to them, or they can be dynamic, i.e. any task can initiate another set of tasks within the component, e.g. a task thread fires an event handler.

3- The task itself *may* be divided into a set of inner sub-tasks that are initialized and executed within scoped memory areas nested in its scoped memory stack.

4- The tasks within the memory model may need to communicate with each other in order to provide the required service.

5- As it is a subcomponent within the component model, the memory model subcomponent should be easy to integrate with the component framework. In other words, it can access and be accessed by other parts of the component model such as the container.

From an RTSJ's viewpoint, we propose that these assumptions can be mapped into the *Forked Memory Model* (shown in Figure 5-16) where this proposed model consists of:

- **A single parent memory area (CMA);** this memory area acts as shared memory area (***root MA***) for all memory area stacks (***leaves MAs***) of the tasks within the component.
- **A set of single memory area stacks (SMA$_{i\ldots n}$);** where each memory area stack SMA$_{x\in[1,n]}$ represents a separate memory area assigned for each inner task *x*, and its nested inner sub-tasks. The scoped memory areas within the stack are nested, where each scoped memory area within the stack is created from its next lower scoped memory in the stack which holds the memory object of it.
- **The Container Memory Area (ContMA);** the component memory area itself is created from within the container memory area, ContMA.

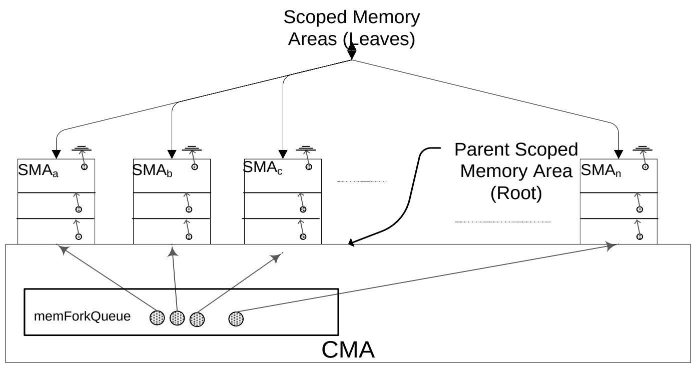

Scoped Memory Areas (Leaves)
SMAa
SMAb
SMAc
Parent Scoped Memory Area (Root)
SMAn
memForkQueue
CMA

Figure 5-16: Basic Component's Forked Memory Model

A- Memory Model Sub-Component Design

In our design of the memory component sub-component, we have proposed that the component is implemented using a class `ForkedMemoryModelCls`, see an example implementation in A.3. The `ForkedMemoryModelCls` implements the interface `IMemoryModel` shown in Figure 5-17, as we assume in the functional component model that it defines a reference to this memory model interface to access its services. We have chosen this way to ensure that the component model is not tightly linked to a single implementation of the memory model, but it can accept different forms of the implementation as long as it implements this interface.

The configuration, initialization and creation of a memory hierarchy within the component according to the proposed model and the above assumptions are explained in the following sections.

B- Memory Configuration

The memory areas used within the memory model of the component has to be configured by the developer. This involves the configuration of both the component's common memory area and the tasks' memory areas. So, in this section we will discuss how these memory areas are configured.

```
package RTCOM;
import javax.realtime.*;
public interface IMemoryModel
{
        public void buildForkedMemory(Class typeOfCMA, long CMA_InitialSize,
        long CMA_MaxSize, int maxSubStacks,Class typeOfSMA,
        long[] SMAs_InitialSizes, long[] SMAs_MaxSizes, Class MPortalType,
        Class SMAsQueueType, Class ObjAllocator, Class MemCtrl);//builds the inner
        //structure of the component, including the SMAs and the sub-components
        public MemoryArea getSMAMemory(String smaName);//Get the memory of a certain SMA
        public MemoryArea getCMA();//Get the Common memory area
        public Object getPortalOfCMA()//Get the portal of the CMA
        public IQueue getSMAsScopes();//Get the SMAQueue
        public IQueue getHeldSMAsScopes();//Get the held SMAs
        public ScopedMemory attachNewSMA(final Class reqMemType,final long initSize,
        final long maxSize);//Attaches a new SMA
        public void setObjectAllocator(IObjectAllocator objectAllocator);//Assign an
        //object allocator to be used
        public IObjectAllocator getObjectAllocator();//Get the Object Allocator
        public void setMemoryController(Class MemoryCtrlCls);//Assigns a memory life
        //time controller
        public IMemoryModelController getMemoryController();//Retrieves the memory
        //lifetime controller
        public ScopedMemory getContainerMA();//Get the container memory area
}
```

Figure 5-17 IMemoryModel interface

Configuration requirements of the parent memory area (CMA/root)

The parent memory area must be externally configurable by the developer during the *component configuration stage* at design time (in statically created systems) or during runtime (in dynamically created systems).

The memory model interface should enable the developer to configure the memory source from which the parent memory would be allocated, according to the required lifetime and requirements of the program using this component; e.g. for non-real-time applications, it can be created from heap memory. Otherwise, for real-time systems, it can be allocated from immortal memory as long as this component is going to be alive for the system lifetime, or from a linear (or variable) time scoped memory area if the component has a shorter lifetime than the program using it. The selection of a certain memory area can be reflected later on using some patterns internally to construct the component's memory model, e.g. the use of portals of the parent memory areas, as explained later, will be restricted to a parent memory area that is configured to be of a scoped memory area type. Hence, we have put a restriction on all the patterns associated with the forked memory model presented in this chapter to by assuming that the parent memory area is of a scoped memory area type, where this restriction does not limit these patterns to be extended to work with other memory area types with some modifications.

Configuration requirements of the child memory areas (leaves)

All the tasks' *leaves* memory areas are by default configured by the component creator within the parent memory area to be of a single subtype of RTSJ scoped memory areas. However, for the dynamically created tasks, the developer should be able to externally configure each one of the added memory areas individually according to the operation specified for each task assigned to them. On the other hand, the developer is not allowed to configure the tasks memory areas to be created neither from the heap memory area nor from the immortal memory areas; this is a basic constraint, as we assume that these memory areas are only for tasks that have lifetimes less than the program in which they are running in and they require predictable memory management.

C- Memory Creation

The method `public void buildForkedMemory (…)` of the `IMemoryModel` interface is responsible of the building of the memory model according to the set of configuration given to it as parameters. These parameters are as follows:

- `Class typeOfCMA`: specifies the class of the type of the CMA memory area.
- `long CMA_InitialSize`: specifies the initial size of the CMA memory area.
- `long CMA_MaxSize`: specifies the maximum size of the CMA memory area.
- `int maxSubStacks`: specifies the initial number of tasks' memory areas.
- `Class typeOfSMA`: specifies the class of the type of all tasks' memory areas.
- `long[] SMAs_InitialSizes`: an array that holds the initial sizes of all the tasks' memory areas.
- `long[] SMAs_MaxSizes`: an array that holds the maximum sizes of all the tasks' memory areas.
- `Class MPortalType`: specifies the class of that implements the multi-named portal.
- `Class SMAsQueueType`: specifies the class that implements the *memoryForkQueue* that holds references to the tasks' memory areas that are currently pinned in memory.
- `Class ObjAllocator`: specifies the class of the memory object allocator to be used by this memory model.
- `Class MemCtrl`: specifies the class of the memory controller that manages this memory model.

The execution of this method is responsible of the creation of the memory model in a way that integrates the model parts together by using our proposed *Multi Named-Object Pattern* in order to enforce the use of RTSJ memory access rules as explained next.

At the initialization stage of the component, its initial/static memory areas are created according to its design time configurations. Firstly, according to the given parameters, the CMA memory area is created from the component's source memory area. Then, the scoped memory area of each inner task is created. The creation of objects representing these scoped memory areas (***leaves***) in our proposed *Forked Memory model* is done by an initialization thread running within the *CMA* memory area (e.g. by the *Manager Thread* of the container holding the this memory sub-component). Also, we assume that the references for these objects are allocated within the *CMA* memory area and all these created references are kept inside a single predefined collection (*memoryForkQueue*) within this CMA memory area.

Adopting this mechanism enforces the RTSJ memory access rule as required in this model, as any real-time thread that wants to access an object within any of the *leaf* scoped memory areas has first to enter or execute in the CMA memory area to be able to get a reference to the required task's scoped memory area from the references collection, i.e. *memoryForkQueue*.

In addition to the static addition of the memory areas, the model also allows dynamic addition of any more task memory areas, either explicitly by using the method `attachNewSMA()` that creates and adds a new task memory area to the *memoryForkQueue* of the specified scoped memory area type and initial and maximum memory sizes, or implicitly by allowing the creation dynamic tasks using the method `createReusableStackTask()` that creates a new task and add it dynamically to the component.

D- Memory Access

Accessing the inner memory areas of the memory structure should be subjects to the memory access laws of the RTSJ, e.g. the CMA can be accessed from the CMA itself or from any inner memory area, where the model provides the following methods to access its inner memory areas:

- `getCMA()` method: retrieves a reference to the CMA memory area.

- `getPortalOfCMA()` method: retrieves a reference to the portal of the CMA memory area.
- The `getSMAScopes ()` retrieves a reference to the `memForkQueue`.
- The `getHeldSMAScopes ()` is provided to retrieve a reference to the `memHeldQueue` which has references to scoped memory areas held by the memory lifetime controller as will be explained later.
- `getSMAMemory()` method: retrieves a reference to a certain task's memory from the `memForkQueue` using the task's identifier.

The implementation of the `getSMAMemory()`is based on the assumption that the required stack memory area is saved in the *memoryForkQueue* which is created in the CMA memory area, so, the *memoryForkQueue* collection can be accessible as a shared object from any thread running within the component. So, according to the RTSJ memory model, it has to be saved as a memory portal object for the memory area in which it is created, i.e. the CMA memory area. However, RTSJ allows the definition of only one single object to be a portal for any scoped memory area and using it. Hence, in order to extend the use of the portal for multiple objects, i.e. objects other than the *memoryForkQueue*, in our proposed model, we assumed a new simple pattern "*Multi-Named Objects Portal*" pattern (MNPORTAL), in which shared objects are given names in order to be accessed by these names through a single portal of an RTSJ scoped memory. This pattern (shown in Figure 5-18) assumes that the portal of the scoped memory is assigned a single object as required by the RTSJ. This shared object, the MNPORTAL in our model, is a hash map object, where each element in this hash map is an object holding both a reference to a shared object created in the memory scope, and a string name acting as a key that can be used to access this reference from the hash map. As seen in Figure 5-18, a simple shared-object naming scheme can be used to ease using it, one suggested scheme is that the name of the object to be saved is the same as its object references and preceded by the prefix "_ref", where the size of the name should not exceed a specified maximum length to bound the memory usage. Hence, according to this proposed pattern, getting a *leaf*-scoped memory is done using the following three predefined steps:

1. Retrieve the object saved as portal of the root memory by calling the method `getPortalOfCMA()` which in turn calls the `CMA.getPortal()`, defined in the `javax.realtime.ScopedMemoryArea` class, and then cast it as `MNPORTAL` object.

2. Retrieve the shared object representing the *memForkQueue* collection from the `MNPortal` object using the proposed method `MNPortal.getObject ("_refMemForkQueue")` and cast the returned object as `memForkQueue` object.
3. Retrieve the reference object of the required scoped memory using its predefined name from the `memForkQueue` object using the method `memForkQueue.getObject("_refRequiredScopedMemory")`.Where the string `"_refRequired ScopedMemory"` is the name of the required scoped memory shared object.

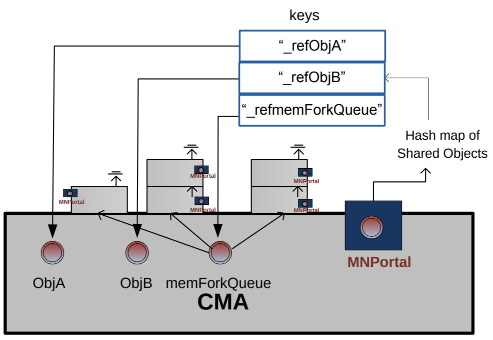


Figure 5-18 The Multi Named-Object Portal Pattern

We have to note that the `getSMAScopes ()` method can be used to replace the first two steps mentioned above. Also, we have to note that the `memForkQueue` holds references to each scoped memory area in the stack that is nested above the CMA, not to any other inner scoped memory areas, where the other nested memory areas above it can be accessed by a chain of calls to the MNPORTAL(s) of this scoped memory area and its inner scopes.

The use of the MNPORTAL is not limited to the CMA memory area, but it is used in all the memory levels of the component model to enable sharing multiple objects. For example, they can be used in the ContMA to share objects among different components that do not share the same CMA memory. Also, in the scoped memory area of the SMA, they are used to enable sharing multiple objects as there is at least one shared object that has to exist in each one of these nested coped memory areas which holds a reference to the next inner scoped memory area in the stack.

5.3.2 Memory Integration with the Component framework

The memory model provides a set of methods that enable the integration with the other parts/subcomponents of the component model. These methods include:

- `public IComponent getComponent():` This method retrieves a reference to the component that holds the component in which this memory structure is located.
- `public ScopedMemory getContainerMA():` This method retrieves a reference to the container of the component that holds the component in which this memory structure is located.
- `public void setObjectAllocator(IObjectAllocator objectAllocator):` This method assigns a certain object allocator to be used by this memory model.
- `public IObjectAllocator getObjectAllocator():` This method retrieves the object allocator assigned to this memory model.
- `public void setMemoryController(Class MemoryCtrlCls):` This method assigns a certain memory controller that manages the life time of the inner memories of this memory model.
- `public IMemoryModelController getMemoryController():` This method retrieves the memory controller of this memory model.

5.4 Object Allocation Management

The memory structure model in our component model, the *Forked Memory Model*, consists mainly of the following memory levels:

- The Tasks' Stack Memory Areas (SMAs) Level
- The Component Memory Area (CMA) Level
- The Container Memory Area (ContMA) Level
- The Immortal Memory Area(IMA) Level
- The Heap Memory Area (HMA) Level

It is a requirement to manage the memory allocation in these levels in order to guarantee the memory required in our framework of the real-time component model. The types and characteristics of these memory areas can affect the allocation management mechanisms that can be applied in the component's memory model. For example, the use of short life time scoped memory for the object allocation areas in the SMAs Level guarantees that the allocated object will be alive only during the lifetime of these scoped memory areas, whereas the allocation of any object in the CMA will enforce this object to be alive during the lifetime of the component itself even if this object is required for only a short period of the component lifetime. In the same way, the allocation of an object in the container itself makes it available for the duration of the container itself even if the component that created it is de-allocated and removed from this container. The same problem exists for the objects allocated in the immortal memory as these objects will never be deleted untill the program that created them exited. On the other hand objects created in the heap can be de-allocated dynamically using the garbage collector. However, using the heap for allocation is not recommended for real-time applications, unless a real-time garbage collector is used, to avoid the unpredictability. So, in order to have a real-time memory model, it is required that the object allocation mechanism to be predictable, i.e. to be bounded by the memory sizes assigned to the memory areas used in the component memory model structure. This can be relatively easy to achieve and control in static components in which it is well known in advance all the objects that are going to be used within them so it is easy to estimate the memory sizes required for the different memory areas in the memory structure of the component. But, in the components that have a dynamic nature of processing, e.g. a server component that can respond to multiple concurrent clients in the same time this cannot be known in advance unless the number of concurrent requests made to this server component is bounded. Even, if the number of concurrent requests is bounded there will be another memory problem, this problem is that the dynamic tasks that are generated within the component to handle the clients' requests may require allocating objects in the CMA or the ContMA or even the IMA Level. After finishing the handling of these requests these objects will be unusable and they will cause a leakage in the component memory which makes the memory model structure of the component unpredictable and makes the component fail to provide its function properly. So it is important in this case to have a mechanism that enable the reusing of these unusable objects from within other dynamic requests to keep the required memory for allocating objects bounded.

Hence, we are assuming that the use of a *reusable objects allocator* is an important feature that has to be available within the component framework to enable the developer to manage the memory allocation particularly in the CMA and the ContMA, and it is recommended to have another one in the IMA as well if required.

This proposed *reusable objects allocator*, as it controls the object allocation, can be built as an optional control sub-component within the framework to be used if the component functionality requires reusability of objects. So, in the following section we will present the proposed structure of this sub-component.

5.4.1 The Design of the Reusable Objects Allocator

In our design of the reusable object allocator, we will borrow the general purpose object recycler pattern presented in (Dibble 2008). See the implementation of this pattern in 0.

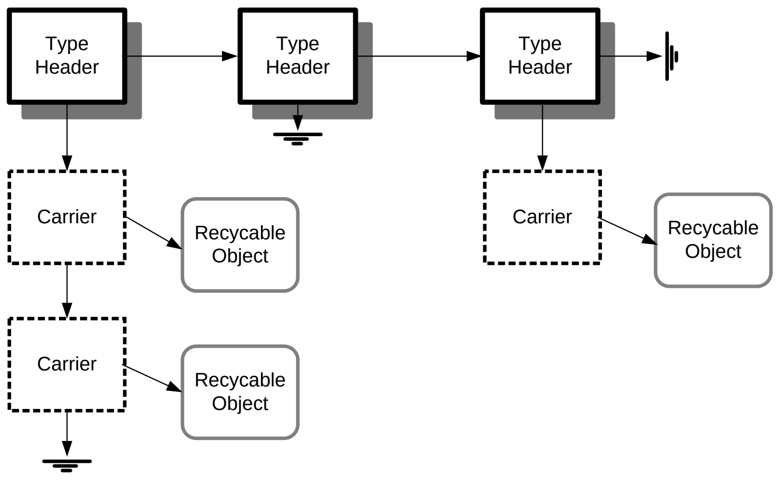


Figure 5-19 Object recycler pattern

As seen in Figure 5-19, this recycler keeps the objects in a list of lists. The first list has an entry for each class of which the recycler has created an instance. Each of those entries is attached to a list of carrier objects that contain references to each recyclable object of that class that is in the recycler's inventory of free objects.

- The interface `IObjectAllocator` of this proposed reusable object allocator consists mainly of three methods:

- `object getInstance(Class type)`. This method is used to get an object instance of the given class name from the list of the free objects of this type in the allocator structure.

- `object getInstanceLike(Object Obj)`. This method is used to get an object instance of the class of the given object name from the list of the free objects of this type in the allocator structure.

- `void recycle(Object unwantedObj)`. This method is used to return an unwanted object, i.e. the object that its current state is not needed any more, back to the list of free objects of its type in order to be reused later.

Due to its internal design, the worst limitation of using the above allocator, that it can works only for objects that have *no-args* constructors. This requires special handling from the developer to encapsulate the classes that have no *no-args* constructors into new classes that have a *no-arg* constructor that creates objects of these classes with a set of default parameters, and at the same time these encapsulating classes should provide accessory methods to control the encapsulated objects change, e.g. assigning the fields' states of the encapsulated objects within them.

As mentioned before we can use instances of this sub-component in the ContMA, CMA, and/or the IMA. This has to be configurable in the component framework during the creation of the component, so there are references of the `IObjectAllocator` in both the `ForkedMemoryModelCls` and in the `ContainerCls,` and there are accessory methods for both of these two classes to optionally create these two reusable object allocators. Where, in general the reusable object allocator created in the CMA provides reusable objects created in this CMA for all objects/components sharing this CMA only, while defining it for the ContMA makes its reusable objects are accessible by all the components and inner objects embedded in this container.

5.5 Memory Model Lifetime Management

In the proposed memory structure for the component, the *Forked Memory Model*, presented above, the memory is composed of a set of nested scoped memory stacks, these memory stacks can be shared among the schedulable objects running within this component or using it. Hence, their lifetimes must be well defined and manageable.

In the RTSJ, a certain scoped memory area is assumed to be valid as long as there is at least one schedulable object running inside it either explicitly using `MemoryArea.enter()` or `MemoryArea.executeInArea()` family of methods, or implicitly as its initial memory area which either assigned to it or it inherits from its parent thread. This model was presented as a memory management model that predictably and dynamically manages memory de-allocation to avoid the unpredictability due to Java's garbage collector. However, this imposes a restriction on using shared objects created within these scoped memory areas. In the RTSJ's memory model the portal object of each scoped memory area instance is used to share objects, where the model assumes that the shared object (or any other object in the memory area) will be valid only as long as a schedulable object is running in it. This has led to the use of the *wedgeThread* pattern presented in (Pizlo, Fox et al. 2004).

A wedge thread is a real-time thread that is created within a scoped memory area, and waits inside it, as long as there is one or more shared objects are to be accessed from this scoped memory area, by a schedulable object which has not yet entered it. The use of this pattern has an overhead, due to the amount of resources needed for the wedge thread. This overhead becomes greater when multiple shared objects exist, and each one of them is created in a different scoped memory area, in this case a wedge thread will be required for each scoped memory scope area to keep it alive (Etienne, Cordry et al. 2006). In future releases of RTSJ, a *pinned* memory model is to be added to the specification.

In our model, we propose that in some cases a certain scoped memory area stack needs to be kept alive, while there is no thread running within it. One example, is when building a component that use the future call pattern. In this case, a thread can create a new scoped memory area to hold the future object and then leaves this scoped memory area to continue other tasks, while another thread, from the same or other component, can enter this scoped memory area later to process the call, and put the result back. Then later, the calling thread can check for the result. This kind of communication is supported in our component model as an inner/inter communication mechanism as mentioned in section -C- in 5.2.6.

From the above, we can see that it is important to have a memory lifetime controller to manage the lifetime of the memory model of the component. This lifetime controller can be built as a control sub-component of the component model that controls the lifecycle of the scoped memory areas that are required to be kept alive.

To build this memory lifetime control sub-component, we can either;

- use the wedge thread model design pattern as a base for the sub-component, an example implementation of the wedge thread pattern, which is compataible with our proposed forked memory model, is presented in A.5., or

- develop enhanced design patterns, which overcome the limitations of the wedge thread pattern.

In the following sections, we first develop a set of enhanced design pattern(s) that can be used to build the required memory lifetime controller sub-component, where these parts have to integrate with the other parts of our proposed component framework. Our proposed enhanced design patterns include two new integrated software patterns, the *forkThread* pattern, and the *dualFork Pattern*. The structure of these patterns and how they work are explained in the next sections.

5.5.1 The ForkThread Pattern

In the *forkThread* pattern, instead of creating a single wedge thread for each scoped memory area that is required to be kept alive, we assume that we use a single thread for *all* scoped memory areas that have a common shared memory (CMA). A simple illustrating diagram for this pattern is shown in Figure 5-20. In the diagram, all the scoped memory areas share the same CMA. Hence, a real-time thread can enter them all at the same time by making a sequence of a pair of `MemoryArea.enter()`, and `MemoryArea.executeInArea()` calls. Then finally, it waits either in the last scoped memory area or in the parent memory area. An example implementation of this pattern is presented in the class `ForkThread`, which is an internal class of the `DualFork` class given in A.6.

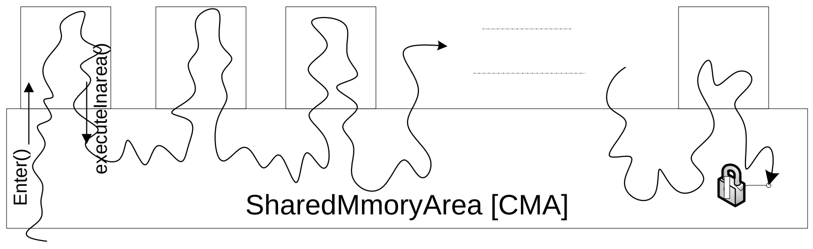


Figure 5-20: ForkThread Pattern

Due to the RTSJ memory access constraints and the nesting required, implementation of this pattern in RTSJ is not a trivial task. Here, we provide our own implementation based on our proposed component memory model presented above. In our proposed implementation, for ease of explanation, we assume that all the scoped memory area stack, which are required to be kept alive, contains only one scoped memory area above the CMA and all these memory areas are saved in another Queue with the name `memHeldQueue` which is similar in the structure to the `memoryforkQueue`, but it contains references only to the scoped memory that have to be kept alive. Also, we assume that the shared objects are accessible through the multi named-object portals defined for each scoped memory area as mentioned before. Figure 5-21 shows a detailed sequence diagram of the pattern.

A- The Propagation Operation

In order to propagate through the scoped memory areas, the following sequence of operations in this pattern is executed:

1- The real-time thread, `forkThread` object, is created and starts in the common memory of the component (CMA), and it waits there waiting for an explicit request from the developer to do a propagate operation into a new scoped memory area.

2- Once a request to propagate arrives, the `forkThread`, retrieves the `MNPortal` object of the common memory.

3- A reference of the `memHeldQueue` is retrieved from the `MNPortal` object. Then, the first scoped memory area is retrieved from the queue.

4- The forkThread starts a recursive propagation process among all the scoped memory areas saved in the queue. The recursion process is provided in a method called `propagate()`, and it includes a repeated sequence of two operations

a. Entering stack scoped memory area using a reference retrieved from the `memHeldQueue`.

b. Executing back into the common memory area, CMA.

5- The implementation of these two operations is done using the *encapsulated runnable* pattern presented in (Pizlo, Fox et al. 2004), where each operation is implemented as a runnable object and then executed by the `forkThread` when it enters (or calls `executeInArea()` of) the appropriate scoped memory area. These two operations are processed by running two encapsulated methods. Hence, as shown in the sequence diagram, they require two nested runnable objects as follows:

- `executeInAreaRunnable`: for executing back in the common memory area.
- `propagateRunnable`: encapsulates the `propagate()` method to enter the next scoped memory area of a certain *leaf* memory area.

6- The above recursive steps are repeated to propagate into any scoped memory area added to the `memHeldQueue`.

7- The Stop-Propagation Operation. This operation is responsible for stopping the recursive propagation operation defined above, i.e. instead of entering the next scoped memory area, the `forkThread` stops propagation at this memory area waiting for external notification. This operation is implemented again using the *encapsulated method* pattern (Pizlo, Fox et al. 2004), where a single runnable, `TailRunnable`, holds the necessary code for causing the `forkThread` to wait at the current (last) scoped memory area.

The recursive nature of the `forkThread` pattern requires careful consideration of the creation of objects to avoid waste of memory resources. `Runnable` objects are the main objects used as they encapsulate the methods that constitute the recursion process mentioned above, these objects are created within the parent memory area, CMA, so they have a life-time equal to the lifetime of the component. This makes the memory leakage very high as each time the fork thread propagate, it creates a new set of runnable objects that are never reclaimed before the component terminates. Hence, to avoid this accumulated memory leakage, we have used the reusable object allocator to create these runnable objects as reusable objects, where each runnable object is recycled once it finishes its execution. An example of this usage is in the next code snippet:

```
IObjectAllocator allocator=(IObjectAllocator)memPortal.getObject("allocator");
executeInAreaRunnable rbranch= (executeInAreaRunnable)
        allocator.getInstance(executeInAreaRunnable.class);
ScopedMemory branch = (ScopedMemory)(item.item());
rbranch.ITEM=item;
branch.enter(rbranch);
allocator.recycle(rbranch);
```

In the above code, taken from the `propagate()` method of the class that implemnts the ForkThread pattern, a reference to the allocator object is retrieved from the MNPORTAL. After that, an instance of the `executeInAreaRunnable` class is allocated by the allocator, then the parameter ITEM of this runnable, is assigned the next scoped memory to be held. Then, the method executed within the `branch` memory area, which represents the current scoped memory area in this iteration. Finally, once the method finishes execution, there will be no need for the runnable object, so it is recycled back to be used in next iteration.

A- Generalization of the Fork Pattern

As said before, the algorithm of the above pattern is working for one level only of nested memory scopes, in order to extend it to nest a full scoped memory area scope stack; we need to a modification to the propagation algorithm as explained next.

The generalization of the fork pattern, see Figure 5-22, requires that the fork thread to propagate to all the nested memory scopes within each stack it enters and execute in the CMA only after it reaches the top scoped memory area in this stack. In our model, this can be achieved by making the thread to check the MNPORTAL of the scoped memory area that is propagated into it to see if there is any non-null reference to inner scoped memory within it or no, if there is an inner one, the forkthread enters it and redoes another check in its MNPORTAL for a next inner scoped memory area, and so on. Once, the forkThread reaches the top of the stack; it can execute in the CMA and continue the propagation algorithm.

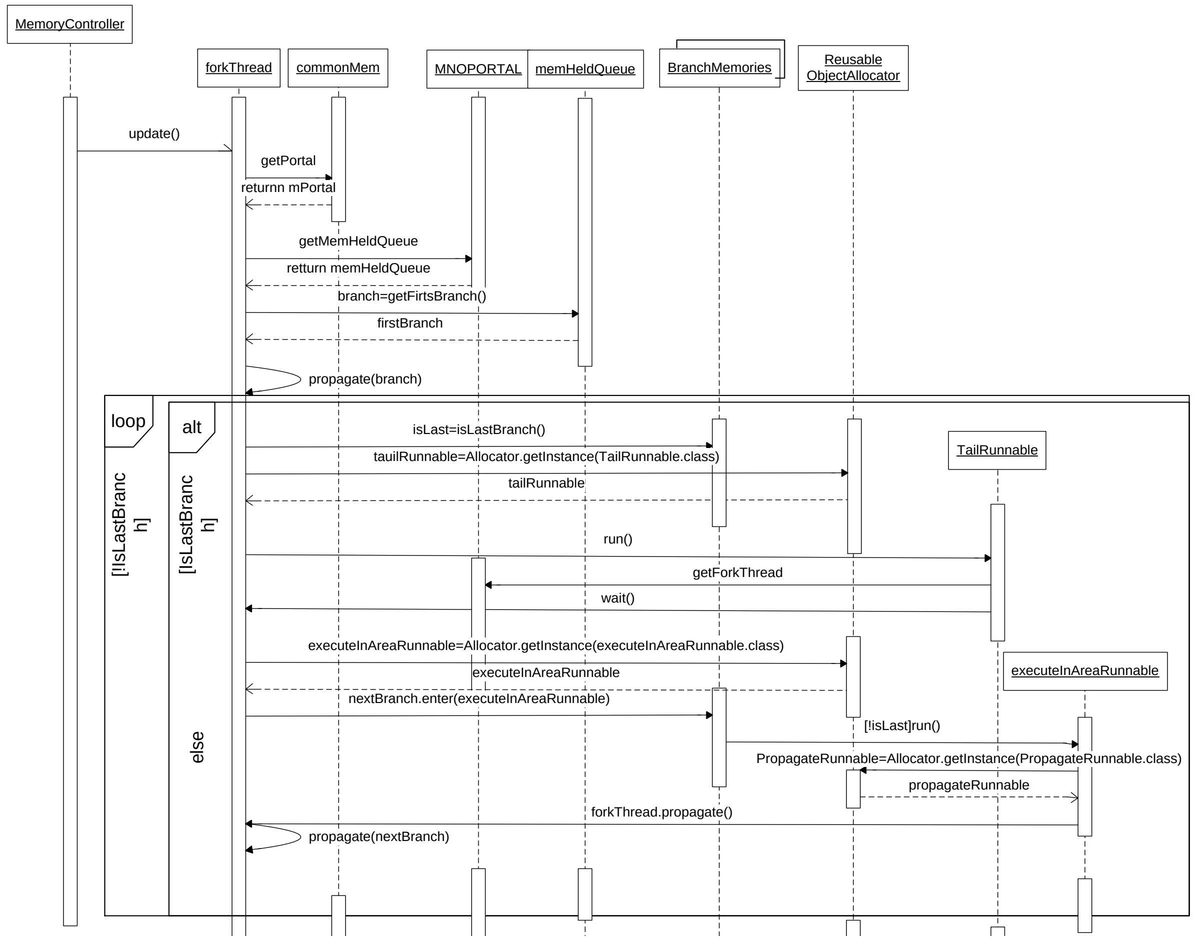


Figure 5-21 ForkThread Sequence Diagram

This modified algorithm requires `executeInAreaRunnable` to support the propagation into the stack, this can be done by changing its `run()` to have the following design pattern:

```
public void executeInAreaRunnable implements Runnable
{
        Public void run()
        {
                MNPORTAL mnportal=getCurrentMemoryArea().getMNPORTAL();
                ScopedMemory sma=mnportal.getObject("nestedMemory");
                if (ma==null)
                {
                        CMA.executeInArea(propagateMethodRunnable)
                        return;
                }
                else
                        ma.enter(this)
        )
}
```

The above design pattern ensures that the fok thread will propagate within all the nested scoped memory areas of the stack before executing back into the CMA memory area.

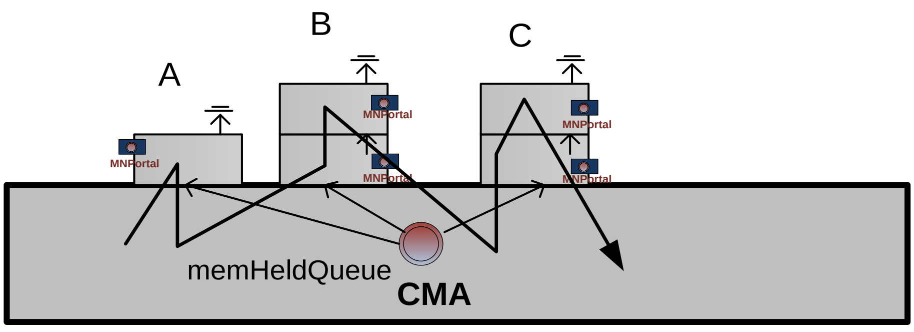


Figure 5-22 General Fork Pattern

B- Limitations of the forkThread pattern

The *forkThread* pattern enables the pinning of a set of RTSJ scoped memory areas however, it is not flexible enough to handle dynamic creation and de-allocation because only two variations of the *forkThread* pattern can be built according to the scoped memories queue associated with it:

A forkThread with a fixed scoped memory list; here neither individual addition nor individual removal of scoped memory areas to/from the `memHeldQueue` is allowed. Hence, applying this model is very restrictive to components that have internal memory stacks with a requirement to be waiting for a certain event (e.g. waiting for other schedulable objects to enter them), and then unpinned when there is no longer need for keeping all of them alive at the same time. Once unpinned, objects in these scoped memory areas can be deallocated; if there is no other schedulable objects running in them. Hence, this model is not suitable for systems with dynamic scoped memory creation, or deletion, which is one of the main purposes of RTSJ scoped memory areas.

One simple use of this form of the forkThread pattern is in building pre-initialized static fork-like memory structure, i.e. a set of inner memory areas are created as inner memory areas of a single shared memory area and kept alive using a fork thread running within them. When the system that usees this assumed memory structure terminates, the fork-thread, can be un-forked to release all the system resources created and associated with this static memory structure. In our framework, this memory model can be used in the container model that is initialized with and holds a fixed set of components. In this model, see Figure 5-23, the container memory area ContMA represents the shared memory area, while the CMA of each component can represent the branches of the fork. These memory areas have to be kept alive from the time of the creation of the components, during the initialization, untill they are all terminated together. This model is valid as long as there is no dynamicity in addition or deletion of the components.

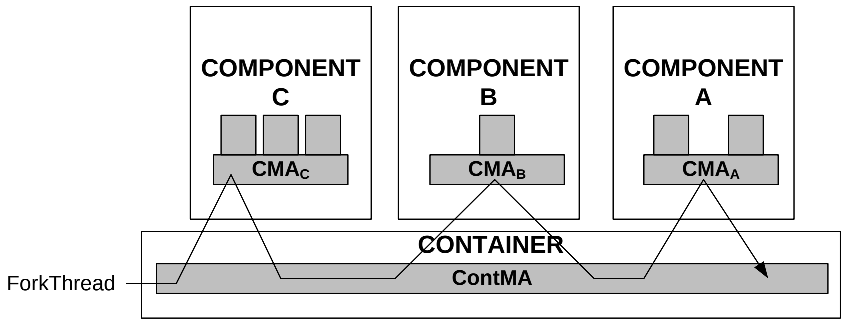


Figure 5-23 Fork thread with fixed scoped memory list

A forkThread with append-only/remove last scoped memory list; this model is a simple extension of the previous one, by allowing the *forkThread* to propagate only within a new memory area(s) appended to the end of its associated scoped memory list, or de-propagate from the last memory area(s) removed from its associated memory list. This model does not add much to the previous model, as it only relaxes the restrictions on a subset of the associated scoped memory areas (i.e. the memory areas appended or removed to/from the tail) not all of them.

An example of possible use of this pattern in our model is in building the memory structure of a static set of components within a container with the ability to dynamically bind an external component(s) to them, see Figure 5-24, where this external component (Component X) can be appended in to the memory areas queues. Then later, when it is not needed, this bound component can be unbounded and de-allocated by de-propagating the fork thread from it.

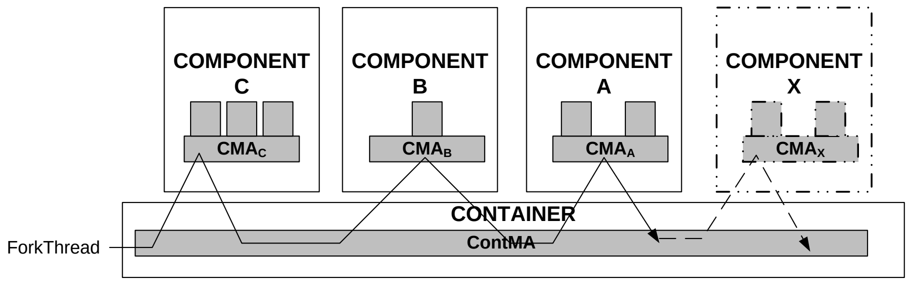


Figure 5-24 Fork thread with append-only/remove-last scoped memory list

The Life Time Controller Implementation

The fork pattern can enable us to build the scoped memory areas life time controller, e.g. by implementing the `ISMALifeController` that supports these patterns:

```
synchronous hold (String SMAname, Scoped Memory SMA)
{
     memHeldQueue.ADD(SMAname ,SMA);
     FORKTHREAD.upDateFork(HOLD_SMA_Queue);
}
synchronous unHold (String SMAname)
{
     memHeldQueue.REMOVE(SMAname);
     FORKTHREAD.upDateFork(HOLD_SMA_Queue);
}
```

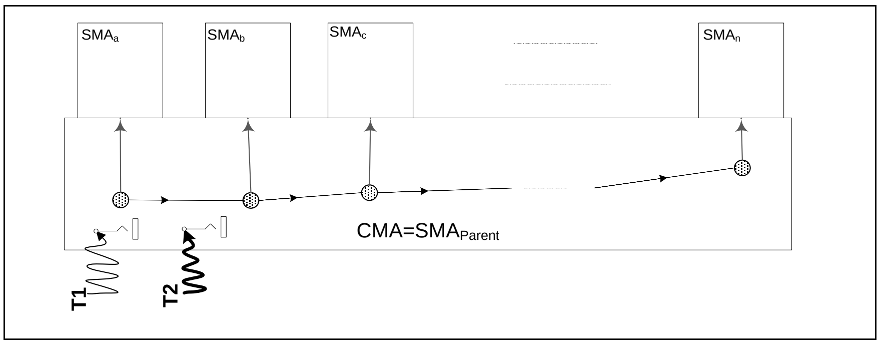
In the above patterns, the `memHeldQueue` represents the named queue that holds all the scoped memory area objects that are currently held.

5.5.2 The DualForkThread Pattern

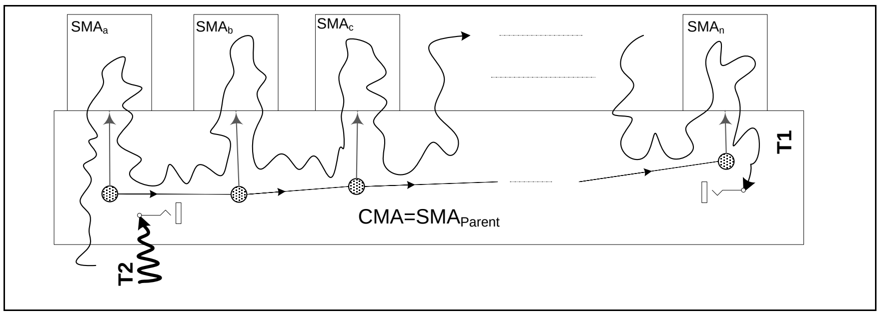
In order to have a more generalized memory life cycler controller model, we need to remove the limitations of the *forkThread* pattern. In order to do that, we propose the *dualForkThread* pattern, see an example RTSJ implementation of this pattern in the `DualFork` class presented in A.6. The *dualForkThread* pattern is formed simply of two individual fork-threads that are running concurrently and cooperatively to achieve the requirement of keeping a set of scoped memory areas sharing a single memory area alive as long as needed, where this set of scoped memory areas is a dynamic set, i.e. inserting new scoped memory areas or removing some existing ones is allowed during the lifetime of the *dualForkThread*. To explain how the *dualForkThread* pattern works, the diagram shown in Figure 5-25[a-h] shows the different states of the two fork-threads (T1, T2). This is explained in the following:

a. Initially, there is no scoped memory areas assigned to the two threads so, they wait in the parent memory area (CMA).
b. Once a scoped memory area set has been created, the first thread T1 propagates among all scoped memory areas defined in it, as explained before for the forkThread pattern, and finally waits at the last scoped memory area, while the other thread T2 is still waiting in the parent memory area.
c. Then, at any time, a new scoped memory area can be inserted anywhere into the scoped memory area set hence, the two threads are required to take actions to update their state.
d. So, the second thread starts to propagate within all the scoped memory areas within the memory areas set including the new one, and finally stops and waits at the last one.
e. Once the second thread arrives at the last scoped memory area, it notifies the first one to de-propagate, i.e. to exit from all scoped memory areas it has entered before and then it stops and waits at the parent memory area.
f. Hence, in this manner, the *dualForkThread* pattern enables dynamically inserting scoped memory area(s) to its associated list, which was not possible with the *forkThread* pattern.
g. Now assume that one of the current scoped memory areas within the memory set is not needed any more and it is required to release it. So, the two threads swap their actions done in steps (c, d, e), i.e. thread T1 propagates within the scoped memory area set, which does not include the removed scoped memory area, then it stops and waits at the last scoped memory area however, before stopping, it notifies T2, to de-propagate back to exit all the scoped memory areas it has entered before,
h. Hence, as the removed scoped memory area has no thread running in it, it can be freed (if it has no more schedulable objects running within it), and the threads wait for new requests for updating its status or for terminating.

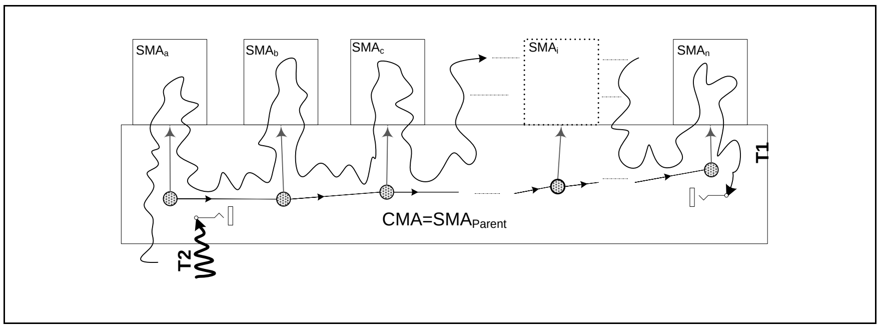
In our design, we assume that the dualForkThread pattern with its associated concurrency control mechanism of the operations explained above is encapsulated into a single control component, where the main function of this component is initializing and managing two fork thread object (T1, T2) as shown in the activity diagram given in Figure 5-27, where T1 and T2 have the same sequence of activities along their lifetime within the component, and each one of them is either in one of four states:



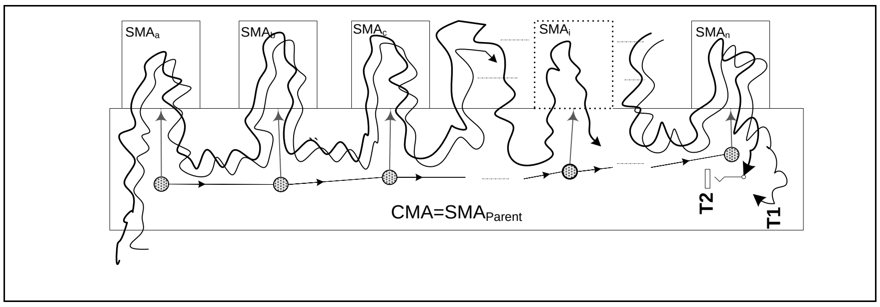
a- Initial State



b- Propagation of Thread T1

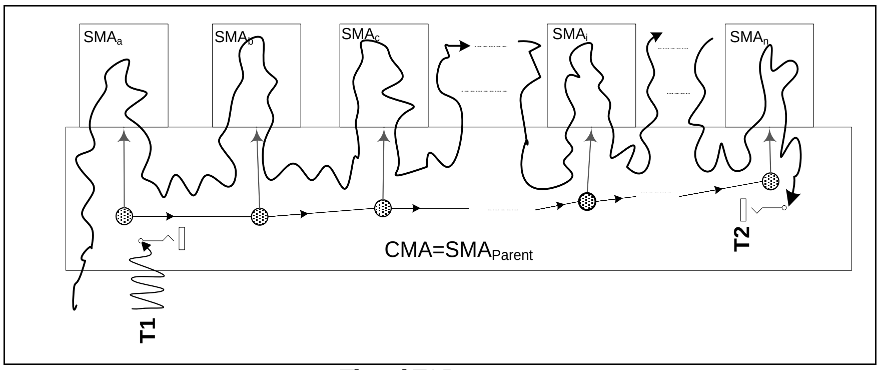


c- new Scoped Memory inserted

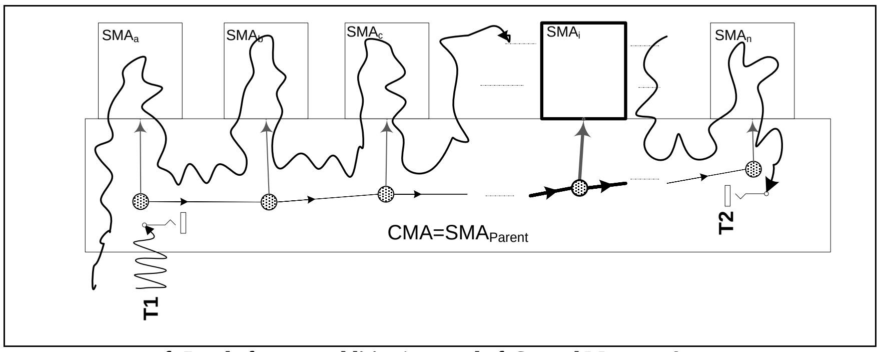


d- Propagation of Thread T2

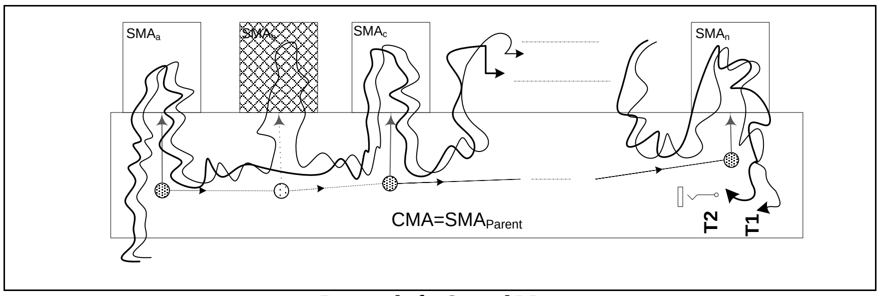
Figure 5-25 DualForkThread States [continued]



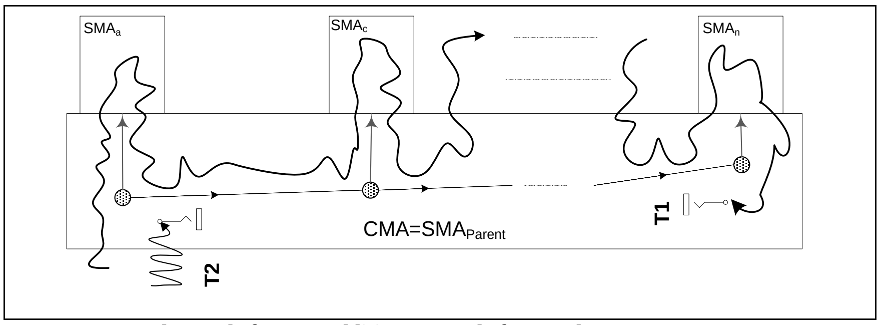
e- Thread T1 Depropagates



f- Ready for next addition/removal of Scoped Memory Areas



g- Removal of a Scoped Memory



h- Ready for next addition/removal of Scoped Memory Areas

Figure 5-26 DualForkThread States

1- **Waiting for a request to propagate.** In this state, the fork-thread is waiting for an update command on a `forkLock` object. The `forkLock` object is a simple object saved in the shared parent memory area, CMA, accessible by both fork-threads. Once an update command is received, a signal is sent to the `forkLock` to notify both fork-threads to start to propagate through the scoped memory areas, if both of the fork-threads are waiting in this state, i.e. no scoped memory areas were associated with the dual-fork threads; then, only one of them is woken, while the other is enforced to wait for the next update operation, the choice of the thread to be woken is simply done by testing a simple primitive value *`turn`* defined in the dual-fork object which alternates its value between 1 and 2 on each update call. However, before the woken thread proceeds, it has to be confirmed that the other thread is not active, i.e. it is not currently propagating or de-propagating, to ensure exclusive operation of each of them. Hence, it has to check if the other thread is in states (b or d). If the other thread is neither in state (b nor d) then, this thread moves directly to state (b). Otherwise, the thread stops again on another shared object `updateLock` waiting for a notification signal from the other thread to inform it that it is safe to continue to move to state (b).

2- **Propagating through the scoped memory area set.** In this state the thread recursively propagates to enter all the current members of its associated scoped memory set. Once the thread enters the last scoped memory area, it becomes safe for the other thread to continue propagation if it is waiting for this thread to finish propagation. So, this thread sends a signal `(updateFork.notifyAll())` to release the other thread.

3- **Stopping in the last scoped memory area.** In this state, the thread stops waiting for the other fork-thread to receive a command to update in order to replace this thread. Once the other fork-thread receives the update command and propagates and reaches the last scoped memory area of the scoped set, it sends a notification signal `(updateLock.notifyAll ())` to this thread to release it and enable it to move to state (d).

4- **De-propagate back to the beginning.** In this state the thread exits from all the scoped memory areas it is currently entering them, i.e. it returns back to its initial state (a). Once it returns to the initial state, it sends a notification signal `(updatLock.notifyAll())` to enable any waiting fork-thread to start to propagate as explained before.

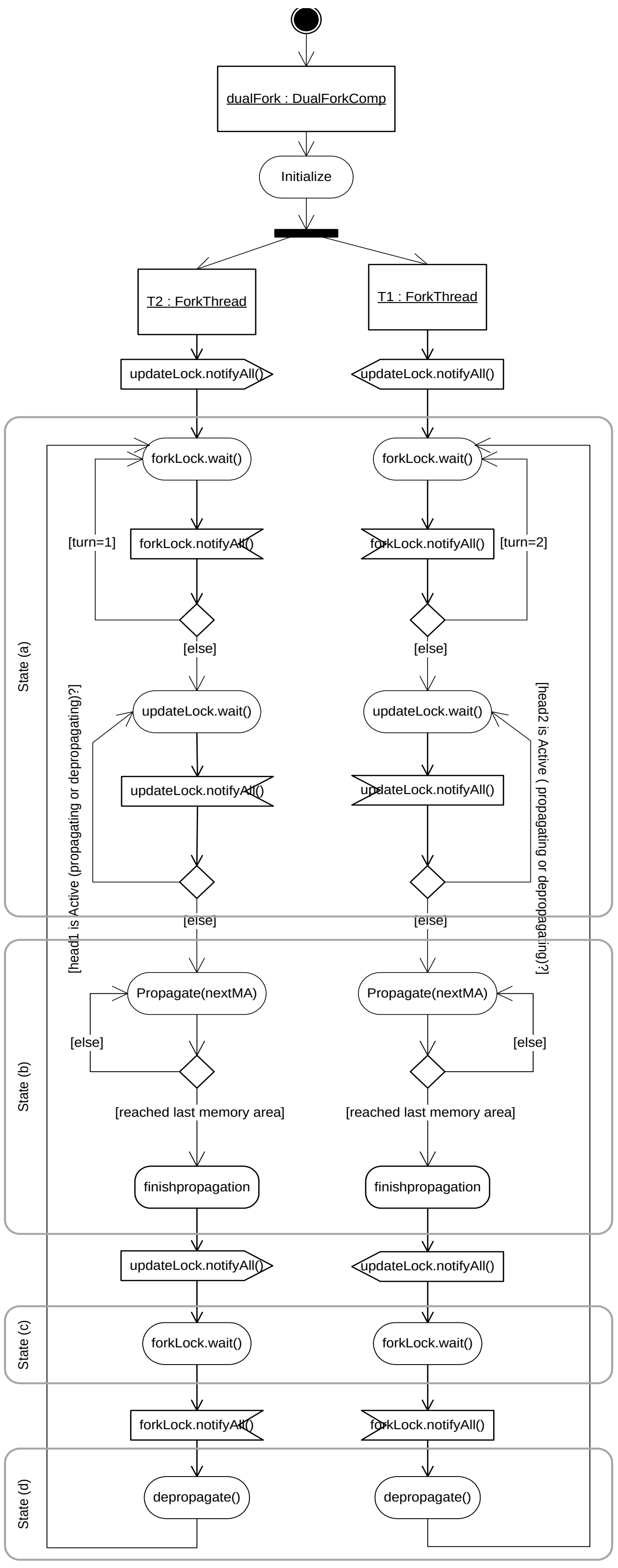


Figure 5-27: Activity Diagram of the DualForkThread Pattern

The Life Time Controller Implementation

The dual fork pattern can enable building a scoped memory areas lifetime controller, e.g. it can be used to build a class that implements an `ISMALifeController` using these patterns.

```
synchronous hold (String SMAname, Scoped Memory SMA)
{
  memHeldQueue.ADD(SMAname ,SMA);
  DUALFORKTHREAD.upDateFork(HOLD_SMA_Queue);
}
synchronous unHold (String SMAname)
{
  memHeldQueue.REMOVE(SMAname);
  DUALFORKTHREAD.upDateFork(HOLD_SMA_Queue);
}
```

In the above patterns, the `memHeldQueue` represents the named queue that holds all the scoped memory area objects that are currently held.

5.5.3 The Memory Pinner Pattern

In RTSJ, the lifetime management of any scoped memory area is managed by counting the number of schedulable objects references running within this memory area, where each schedulable object enters the memory area increases a reference count variable, and this reference count variable is decremented each time once this schedulable object finishes executing its logic in this scoped memory area. Once, the reference count goes to zero, i.e. there is no more schedulable object running within this scoped memory area, all the objects created within this memory area are reclaimed.

In the current RTSJ, the user has no explicit access to the reference count variable; so, he cannot manage to prevent the reclaim process once the last schedulable object exits the memory area. This enforces the user to keep at least one schedulable object running within the scoped memory area to keep it alive.

The Life Time Controller Implementation

As seen, the above methods enable the user to explicitly manage the life time of the scoped memory areas. So, we can build our proposed scoped memory life controller using a memory pinner controller that implements the `ISMALifeController` using these patterns.

```
synchronous hold (String SMAname, Scoped Memory SMA)
{
        memHeldQueue.ADD(SMA);
        SMA.PIN()
}
synchronous unHold (String SMAname)
{
        ScopedMemoru SMA= memHeldQueue.REMOVE(SMAname);
```

```
            SMA.UnPIN()
    }
```

In the above patterns, the `memHeldQueue` represents the named queue that holds all the scoped memory area objects that are currently pinned.

5.6 Reusability of Schedulable Objects

In our component framework model, we assumed that the inner tasks of the component are executed by schedulable objects. The component can have a fixed set of schedulable objects that are executing its inner tasks during the lifetime of the component; in this case instances of the RTSJ's schedulable objects can be assigned during the initialization of the component to execute its internal tasks as required. However, in the case of a component that dynamically creates inner task, e.g. a server component that creates handlers to handle the client requests, which means that these dynamically created tasks needs the dynamic creation of schedulable objects to execute them, where these schedulable objects will be required only for the duration of the execution of their tasks, e.g. in the case of the server component the schedulable object that replies to the client request is required only for the duration of processing the client request and it is not needed afterwards. As these dynamically created schedulable objects are created within the memory structure model of the component, so they are as any other object can cause memory leakage if their allocation is not controlled and bounded. In addition to that, as the number of concurrent existing instances of these schedulable objects determines the level of the concurrency of the component. So, as a requirement of many real-time systems, there is a requirement to limit this number to guarantee a predictable level of service. Hence, the component framework must present a mechanism to enable the reuse of the schedulable objects within it. This mechanism should support the following:

1. The creation of schedulable objects pools of a maximum size of a certain required schedulable object type.
2. The presentation of reusable forms of the schedulable objects that can be inserted in these schedulable object pools.
3. As the schedulable objects are executing the task's logic defined by the users, there is a need to separate the logic from the schedulable object, so that the same reusable schedulable object can be reused in executing other task's logic when it is recycled.
4. As the task logic is assigned by the developer, the separated representation of the task's logic must be in a simplified and abstracted form that enables the execution

of the logic along the different layers of the memory model without the complication of the RTSJ memory model.

To achieve these requirements we assume that the component should have *optionally* the following:

1. Reusable schedulable objects Pools as Structural sub-component.
2. Reusable schedulable objects, or Executors, as Structural sub-components that can be saved within these pools.
3. Reusable Logic sub-component that can be assigned by the developer to be executed by the reusable schedulable objects.

We will present our design of these three sub-components in the following sections:

5.6.1 Pools of Reusable Schedulable Objects

We assume in our design that the pool of any schedulable objects implements the `IHandlerPool`, where this interface has the following methods:

- `public IHandler getFreeHandler().`To get the next free executor of the poll with the given minimum memory size.

- `public IHandler getFreeHandler(long reqMemSize).` To get a free executor of the poll with the given minimum memory size.

- `public void appendHandler(IHandler executor).` To add either a new or recyclable executor to the pool.

- `public int getSize().` To get the total size of the pool.

- `public int getFreeCount().` To get the size of the current free executors in the pool.

The class `HandlerPool`, which is given in A.7, provides an example implementation of the `IHandler` interface, this class is using a queue that is filled, during the initialization of the component, with the required maximum number of reusable schedulable object instances of the required type.

5.6.2 Reusable Schedulable Objects Sub-Component

The reusable objects in our model are implementing the interface `IReusableExeutor` which defines the following methods:

- `public void start(): to start the execution of this reusable schedulable object.`

- `public long getMaxSize():`to get the maximum size of the memory area assigned to this executor.

- `public MemoryArea getExecutorMemoryArea():` to get the memory area assigned to this schedulable object.
- `public SchedulingParameters getSchedulingParameters():` to get the currently assigned scheduling parameters of the executor.
- `public ReleaseParameters getReleaseParameters():` to get the currently assigned release parameters of the executor.
- `public Runnable getHandlerLogic():`to get the current logic to assigned to this handler.
- `public void setSchedulingParameters(SchedulingParameters schParamas):` to change the currently assigned scheduling parameters of the executor.
- `public void setReleaseParameters (ReleaseParameters relParamas):` to change the currently assigned release parameters of the executor.
- `public void setHandlerLogic(Runnable executorLogic):` to assign the logic to be executed by this handler. This method is important for reusability of the schedulable object as it enables to change the logic that it runs.

It must be noted that the last three methods are important methods as they are responsible of updating the execution parameters of the executor to be reusable.

As there are different types of schedulable objects, we need different classes to implement the above interface. Where each of these classes represents one type of schedulable objects defined in the RTSJ. These classes should have the ability to offer the last three methods in the `IReusableExeutor` mentioned above, i.e. changing the logic, the scheduling and the release parameters. The schedulable objects in the RTSJ already offers the methods that enable changing the scheduling and the release parameters but it does not offer any method that enable the change of the logic run by these schedulable objects. To solve this problem, we use the design patterns defined in (Dibble 2008) to support reusability of schedulable objects. These patterns are:

1- Encapsulation of the schedulable object in a wrapper class that implements the schedulable interface. This method is applicable to real-time threads, where a wrapper class is created with an instance of the `RealtimeThread` class, where this wrapper class implements the methods of the schedulable interface by forwarding them to the inner RealtimeThread instance, with a special handling for the `run()` method, where the constructor of the wrapper class creates an instance of the inner thread with its runnable logic is the wrapper class instance itself, this to force the inner realtime thread to execute the wrapper class's run method. While in the run method of the wrapper class a code similar to the code pattern in Figure 5-28 is inserted.

This pattern enforces the inner method to run the run method of the `assignedLogic object`, which is a member variable of the reusable thread class that represents the runnable object that can be assigned when reusing this reusable real-time thread.

2- Extending the class of the schedulable object. This method is more suitable for the Asynchronous Event Handlers, because the AEH's `handleAsyncEvent()` method or its `run`() method is executed repeatedly each time an event is fired. So, it is easier to make it reusable in a new class that extends the AsyncEventHandler class and overrides its method `handleAsyncEvent()` to run the pattern shown in Figure 5-29.

```
while(keepAlive)
{
       try
       {
         assignedLogic.run();
       }
       catch (Exception e){ }
}
```

Figure 5-28 pattern used in run() method of reschedulable object.

```
if(assignedLogic!=null)
  assignedLogic.run();
```

Figure 5-29 code pattern used in the handleAsyncEvent()

This enforces the AEH to run the logic defined by the `assignedLogic` object, which is a member variable of the reusable AEH class that represents the runnable object that can be assigned when reusing this reusable AEH.

In our component framework, we assume that all the pools that hold the reusable schedulable objects are kept in the container that acts as the executors' supplier for all the components that reside in this container. Hence, according to this assumption and in order to satisfy the RTSJ memory access rules, all of the reusable object instances have to be created within the container memory area ContMA so that these references can be saved in the schedulable objects pool(s).

5.6.3 Reusable Stack Logic Sub-Components

As mentioned in the previous section the reusable schedulable objects have to be created within the container's memory area. So, we assume that all the schedulable objects are initialized from the container memory area and each schedulable object can have its own stack that has the container memory area ContMA and the component memory area CMA as the lowest memory areas in this stack, see Figure 5-30.

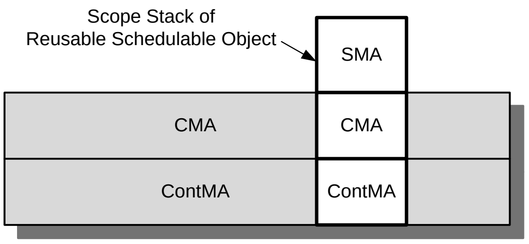


Figure 5-30 the task's scoped memory stack

According to this model the reusable schedulable object has to start its execution in the ContMA, however, according to our framework model, the schedulable objects of a certain component has to execute in the CMA and the SMA memory areas of this component, where the CMA should be used for communication with other schedulable objects within the same component and SMA, and any scoped memory nested within it, can be used for operations that require temporary objects. Hence, the developer has to write his own code that propagates to the CMA then the SMA to execute his code there.

In order to simplify this and make it less complicated, we assume the Executable Runnable Stack design pattern. In this new design pattern, we assume the following, see Figure 5-31:

1. we have *n* of nested scoped memory {SMA0, SMA1, …, SMAn} areas within a single stack, and
2. each scoped memory area SMAj is nested within in another scoped memory area SMAi where $j=i+1$and this memory area SMAj is created from within SMAi which stores a reference for it in its multi-named portal object portal, MNPORTAL, defined earlier in this chapter.
3. It is assumed that one or more of the scoped memory areas in the stack are pre-existing, i.e. the deep most scoped memory areas exist, while the inner ones are created dynamically when the next operation in sequence does not have its own memory area in the stack.
4. there is a set DOPS of *2n* sequential tasks that have to be executed within these memory areas in sequence using a single schedulable object, where
5. Only two operations are executed in each single memory area, one when the schedulable object enters the memory area and the other one is executed when it leaves the memory area.

6. each operation can be dependent on the previous operation(s) i.e.
7. DOPS={OP-UP0,OP-UP1, .., OP-UPn, OP-DNn,OP-DNn-1 , …., OP-DN0}
8. Also, we assume that the first sequence of these operations OP-UP0=>OP-UP1=> …=> OP-UPn are executing in a bottom-up approach in the memory areas of the stack when the schedulable object enters them, while the OP-DNn=>OP-DNn-1 => …., OP-DN0 are following them and executing in each scoped memory of the stack before leaving it in a top-down approach.
9. The logics of all the operations are to be supplied by a single file by the developer which hides the details of entering and leaving the memory areas.

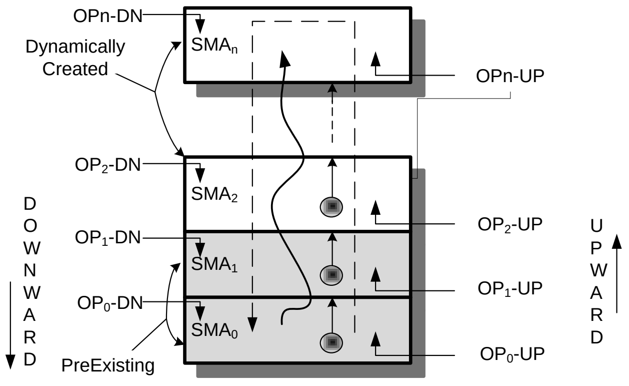


Figure 5-31 The Runnable Stack Pattern

According to the above assumptions we can build the reusable stack as a class that implements the Runnable interface in order to be executed by the schedulable object, while the `run()` method is implemented using the recursive algorithm shown in Figure 5-32, An example implementation of this class is presented in the class given in A.7.

```
1- If(Current Memory Area Level above or the same Existing Memory Area Levels
and less than the maximum number of nested memory levels)
   - Create a new nested Memory Area M
   - add a reference "ChildMA" to M in the MNPORTAL of the current memory area
2- Call AssignedStackLogic.runUpward(Current Level Number, current Component)
3- Enter the memory area of the next level
4- Call AssignedStackLogic.runDownward(Previous Level Number, current
Component)
```

Figure 5-32 Algorithm for the run() of the Runnable Stack

The algorithm shown above recursively propagates in the scoped memory stack where it creates the required nested scopes memory areas if they do not exist already, and each time it enters a scoped memory area it calls the `runUpward()` method of the object, where it passes to it the current memory level, and a reference to the component in which the schedulable object that executes this runnable object is running. In our model we assume that the `AssignedStackLogic` object is assigned by the developer and it holds logic(s) of the individual operations that are required to be executed within the scoped memory areas. The `AssignedStackLogic` object is assumed to be created by the developer from a class that implements the interface `IStackLogic` shown in Figure 5-33.

```
package RTCOM;
public interface IStackLogic
{
//The following method executes during the propagation into the memory areas of the stack
    void runUpward(int CurrentMemoryLevel, IComponent currentComponent);
//The following method executes during exiting into the memory areas of the stack
    void runDownward(int CurrentMemoryLevel, IComponent currentComponent);
}
```

Figure 5-33 IStackLogic interface

The `IStackLogic` interface defines two methods:

`void runUpward(…)`: this method is to be called each time the schedulable object enters one of the scoped memory areas of the stack, it has two parameters that help the developer to manage his code. The first one is the current memory level within the stack, the second one is a reference to the component itself, so he can access any of the component's elements.

`void runDownward(…)`:this method is to be called each time the schedulable object is about to leave one of the scoped memory areas of the stack, and it has the same parameters as the `runUpward(…)` method.

According to this model, the complexity of entering the memory areas within the stack become hidden and abstracted, as all what the developer needs to write is a class that implements the `IStackLogic` and looks like the pattern shown in Figure 5-34, if applied within our component framework.

5.7 Memory Contracts

In our component model, we proposed that the logics of the tasks are separated from the reusable schedulable objects executing these tasks, this separation is important for the memory management of the component model, as this allows us to define two memory states of the component model:

- The Zero or No-Logic Memory state: this is the memory status that results from the sub-component composing this model, i.e. Object pools, schedulable pools, etc. This state can be analyzed for the component during its design time and can be provided for the component developers to consider it when using the component.
- The With-Logic Memory state: this is the memory status of the component after the developer inserts his own logic to be run using this component. This memory status includes the Zero Memory state in addition to the memory required by the logic provided by the developer.

```
package RTCOM;
public class TaskLogic implemnets IStackLogic
{
    void runUpward(int CurrentMemoryLevel, IComponent currentComponent)
    {
          switch(CurrentMemoryLevel)
          {
                     case 0://Code to be executed when entering the ContainerMA
                     {
                         //Code to be executed when entering the ContMA
                     }
                     Case 1://Component MA
                     {
                         //Code to be executed when entering the CMA
                     }
                     Case 2://Task's SMA
                     {
                         //Code to be executed when entering the SMA
                     }

          }
    }
    void runDownward(int CurrentMemoryLevel, IComponent currentComponent);
    {
          switch(CurrentMemoryLevel)
          {
                     case 0://Code to be executed when entering the ContainerMA
                     {
                         //Code to be executed when leaving the ContMA
                     }
                     Case 1://Component MA
                     {
                         //Code to be executed when leaving the CMA
                     }
                     Case 2://Task's SMA
                     {
                         //Code to be executed when leaving the SMA
                     }

          }

    }
  }
```

Figure 5-34 A component class implementing the IStackLogic interface

In our component model, we assume that the developer has to include with the logic that is required for execution within the component, the memory requirements for this logic in a form of a contract supplied with the logic that specify the memory requirements in the ContMA, CMA, and immortal memory area as well. In order to provide these values, we proposed the interface `IForkedMemoryContract` that has the following methods:

`public long getCMA()`:the developer specifies in it the worst case of memory size consumed by the logic in the CMA memory area.

`public long getContMA()`: the developer specifies in it the worst case of memory size consumed by the logic in the ContMA memory area.

`public long getIMA()`:the developer specifies in it the worst case of memory size consumed by the logic in the immortal memory area.

In order to use this memory contract, we assumed that the developer can present the logic of the task in a class that extends the `IForkedMemoryContract` interface in addition to the `IStackLogic` interface resented earlier.

The importance of using the `IForkedMemoryContract` that it enables the check of the ability of the container to accept the given logic to run within it or no, where the container can provide an admission control checker that checks this contract for all the components added to it, where this check can be done:

- **Statically Configuration/initialization time**: by testing the memory requirements of all the added components in addition to the memory required for the Zero Memory state, to check if the current memory sizes within the component satisfy these requirements or not before starting to run the components.
- **Dynamically at task/component insertion time**: to ensure that the memory requirements of this added task or component is satisfied by the current forked memory sub-component and it will not affect the predictability of any other component running within this container.

5.8 Example: The Method Invoker

In order to see how the different patterns and subcomponents of our framework can be integrated together to support the inter communkication among components, we present here an example that shows how the hand-off pattern can be integrated with the proposed memory model and its associated patterns to make calls from a component to another, where the caller and the called components are examples of the component model provided in the framework.

The hand-off pattern, see Figure 5-35, enables the communication between two sibling memory areas, by enabling one object, the source, in a certain memory area to be accessed by another object, the target, in another memory area, where these two memory areas have at least a single shared memory are, parentMA. The basic structure of this pattern requires, for each communication operation, the execution of two joined objects (BH); **B**ridge, to jump through the parent memory area, and **H**andoff, to deliver the data; generalization of these classes is explained next.

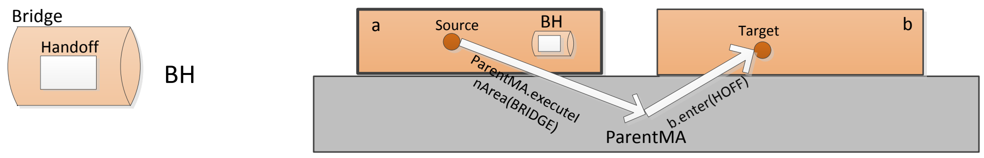


Figure 5-35 The Hand-Off Pattern

In our example, see Figure 5-36, to use the handoff pattern for method invocation, we assume that the called component has the target object saved in one of the scoped memory areas forming the scoped memory stack of this component, and a reference to this target object is stored in the multi named-object portal of this scoped memory area. On the client side, the caller component has a single task running within it, where the call to the target object is initiated by this task. In this example, we assume that each of the caller and the called components has its own stack of scoped memory areas, where the container memory area is the lowest memory area, i.e. level [0] of each of these tasks' memory stacks.

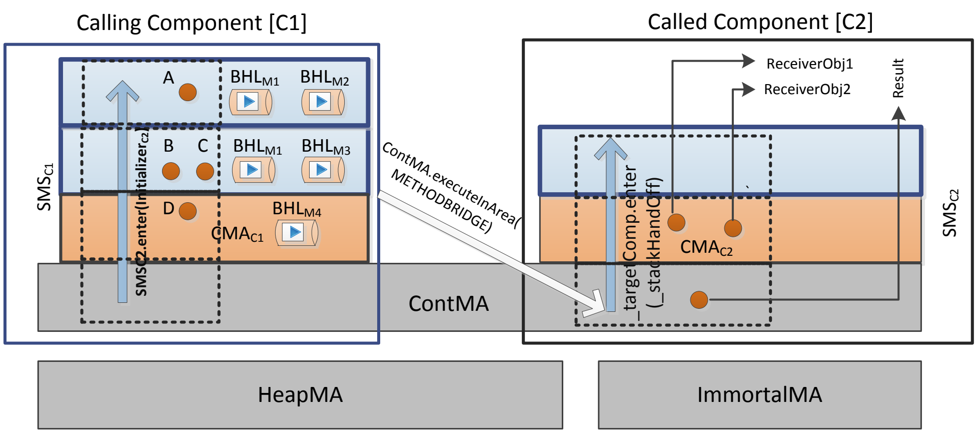


Figure 5-36 The Mehod Invoker Example

This memory structure enables the communication between any two objects residing in the two memory areas stacks of two different components sharing the same container memory area. In this example, we see how we can use our framework to build a general pattern of making calls between two components using this memory structure; the implementation of this general pattern is made using the `MethodBridge` class that provides, in addition to the call invocation methods, the bridge from the caller component to the called component. The `MethodBridge` class implements the `Runnable` interface, and it has an inner class `StackHandOff` that extends the `ReusableRunnableStack` class that execute the handoff on the scoped memory stack of the target component. The `StackHandOff` class itself runs the logic provided by its inner class `MethodCallLogic` that implements IStackLogic interface, to make the logic executable on the scoped memory stack of the target component. In the following, we present these classes.

The `MethodBridge` class: This class is the front-end of the pattern; it provides the methods required to make a call from the calling component to another component. This class has the following key methods:

`makeCall()`. This is a static method, and it is the method that is used to make the call using this pattern, the implementation of this method is shown next.

```
static Object makeCall(IComponent caller, IComponent calledComp, final
Object [] MethodArgs, int MethodIndex,
     final int targetLevel, String tObjName) {
  IObjectAllocator allocator =
      caller.getMemModel().getContainerObjectAllocator(); //Get the object
allocator of the caller
  MethodBridge bridge =
(MethodBridge)allocator.getInstance(MethodBridge.class);
  bridge.setParameters(caller, calledComp); //create an object for the
future call
  Object result = bridge._makeCall(MethodArgs, MethodIndex, targetLevel,
tObjName); //calls the call internally
  return result; //this object has to be recycled by the caller once it
finishes using it
 }
```

As seen abovem the implementation of this method starts with retrieving the reusable object allocator; then, an instance of the `MethodBridge` class is retrieved, using the allocator, from the objects pool to be used to process the required call. This instance is initiated with references of the two communicating components. Then, the call is forwarded to the internal `_makeCall()` method to process it, using the following parameters:

`MethodArgs`: represents the object passed to the target component from the source component. This can hold for example method arguments to the target method.

MethodIndex: This is an ID of the required method of the target object; this ID is used in this pattern to avoid using the reflection to get the required method.

taregtLevel: this defines the memory level of the scoped memory stack in the target component, in which the handoff operation is processed.

tObjName: The name of the target object in the target component.

_makeCall(): This is an internal method that starts the processing of the method invocation, the logic of this method is shown next.

```
 Object _makeCall(final Object [] methodArgs, int methodIndex, final int
targetLevel, String tObjName) {
//In this implementation, we pass args by ref; the method args have to
implement the ICloneable interface if passed by value and they have to be
cloned in the doMethodCall
this.getStackHandOff().useLogic(StackHandOff.MethodCallLogic.class);
     //reset the parameters
  this.getStackHandOff().handoff(methodArgs, methodIndex, tObjName,
targetLevel); //deliver parameters
  return this.getStackHandOff().Result; //this object should be saved in
the container
 }
}
```

As shown above, the logic of this method starts by assigning MethodCallLogic class as the stack logic subcomponent that is to be executed during the handoff operation; then, the handoff()method of the associated StackHAndOff object is executed with the given parameters to handoff the required data to execute the call; these data include the arguments passed to the required method, its index, and the object holding it, in addition to the memory level of the target component, in which the call has to be executed. Finally, after the call is executed, the result of the call is returned to the caller, where this result is stored in the container memory area by the handoff()method.

run(): This method is called by the handoff() method from the container memory area, i.e. parent memory area of the communicating components, during the handoff operation to transfer the execution of the current calling thread from the container memory area to the scoped memory stack of the target component using the statement:

```
  _targetComp.enter(_stackHandOff);
```

The `StackHandOff` class**:** This class extends the `ReusableRunnableStack` class to encapsulate the hand-off operation and make it executable on the scoped memory stack of the target component. This class defines the following:

`_targetName:` the name of the target object

`memLevel:` the memory level in the target component in which the invoked method is executed.

`_methodArgs[]:` this is an array holding the arguments passed to the target method.

`Result:` A reference to the object returned as a result of the execution of the required method at the target component, where this object has to be allocated in the container memory area, in order to be accessible from within the calling component. This object is allocated using the reusable objects allocator in order to be recycled by the calling component, once it is no longer required.

`slogic`: This is an instance of a class implementing the `IStackLogic` class that defines the stack logic which is executed within the target component to execute the required operation at the assigned target memory level of the target component. In this example, this logic is assigned to be of the `MethodCallLogic` class to execute the required method, as explained later.

In addition to the above, the `StackHandOff` class defines the following key methods:

`useLogic()`: This method is used to assign the slogic member of this class, which is the stack logic sub-component that is required to be executed by this class. In this method, see the following code, the existing object referenced by the slogic reference is recycled first, and then a new instance of the inner class holding the required logic, e.g. the `MethodCallLogic` class in our example, is created and then assigned.

```
public void useLogic(Class LogicCls) {
   IObjectAllocator allocator =
   MethodBridge.this.caller.getMemModel().getContainerObjectAllocator(); //
   if (slogic != null)
    allocator.recycle(slogic);
   if (LogicCls.isAssignableFrom(HandOffLogic.class)) {
    slogic = this.new HandOffLogic();//we may use the allocator to create
the stack logic object to avoid memory leaks
   } else if (LogicCls.isAssignableFrom(MethodCallLogic.class)) {
    slogic = this.new MethodCallLogic(); //create the stack logic object
   } else {
    slogic = (IStackLogic)LogicCls.newInstance();
   }
   setStackLogic(slogic); //set the ref of stack logic in this class
  }
```

`handoff()`: This method, see the following code, initiates and starts the method call/handfof operation, it initializes the parameters of this class that encapsulates the handoff process of the target method; then, it executes the logic of the target method by calling the `executeInArea()` method of the container memory area to execute the logic defined in the `run()` method of the MethodBridge, as explained earlier.

```
public Object handoff(Object args [], int mIndex, String targetName, int
memLevel) {
       _targetName = targetName;//set the target name
       _memLevel = memLevel;//set the memory targetLevel of the obj
       backward = false;//set the handoff direction
       _methodArgs = args;//set the source
       _methodIndex = mIndex;
       _targetComp.getMemModel().getContainerMA().executeInArea(
              MethodBridge.this); //executes the call
  }
```

The logic of executing the required call in this class is defined in the inner class `MethodCallLogic`, which extends the `IStackLogic` class, to support the multi-level scoped memory architecture of the component model in our framework. In the `MethodCallLogic` class, the logic for executing the method call operation is defined in the `runUpWard()` method of this class as shown next.

```
   public void runUpWard(int curLevel, IComponent parentComponent) {
    if (curLevel == StackHandOff.this._memLevel) {
     StackHandOff.this.doMethodCall(parentComponent); //execute the handoff
pattern
    }
   }
```

In the above code, the `doMethodCall()` method is executed when the calling thread enters the memory level _memLevel of the `StackHandOff` class, which is assigned by the developer, when calling this pattern.

`doMethodCall()`. This is the key method in this class; it is called by the `handoff()` method from within the memory level of the called component as specified by the caller, it is responsible of executing the required method call of the target object in the scoped memory stack of the target component, the logic of this method starts by retrieving the named portal of the current memory area, which is the memory area specified when this method is called, then the target object is retrieved from this portal using its name, after that the required target object is retrieved from this portal, and then the method index of the required method is used to identify the required method of the target object, then this method is executed. In the implementation of our example, shown next, we assume that one of two target object; `ReceiverObj1`, `ReceiverObj2`, can be specified by the user, where the `ReceiverObj1` has three methods `process0()`, process1(), process2() with the method indices 0,1,2 respectively; whereas, the ReceiverObj2 has only two methods run0(), run1() with the method indices 0,1 respectively. We have to note here that method call is made here by passing the references of the parameters, where accessing these parameters that exist in the sibling memory area of the calling component is allowed in the hand-off pattern. Finally, after executing the required method, the returned value of the executed method, if there is a one, is assigned to the Result reference, where the returned object has to be a recyclable object from the container object pools, to be accessible by the caller, which, after it finishes using it, is responsible of recycling it back to the object pool.

```
  private void doMethodCall(IComponent parentComponent) {
   //thismethod has to be provided by the user of the method implementer/
user
Object R;

   LTMemory curMem =
(LTMemory)RealtimeThread.getCurrentMemoryArea();                //Get
current memory area
   INamedObjectCollection namedPortal =
(INamedObjectCollection)curMem.getPortal(); //Get the MNOP portal
   if (_targetName == "ReceiverObj1") {
    Object tgt = (Object)namedPortal.getObject(_targetName); //retrieve the
target by its name from  the named portal

    if (_methodIndex == 0) {
     R=ReceiverObj1.process0(_methodArgs[0], _methodArgs[1]);  //note no
need to clone the args as they are
    }

    if (_methodIndex == 1) {
     R=ReceiverObj1.process1(_methodArgs[0]);
    }

    if (_methodIndex == 2) {
     R=ReceiverObj1.process2(_methodArgs[0], _methodArgs[1],
_methodArgs[2]);
    }
   }

   if (_targetName == "ReceiverObj2") {
    Object tgt = (Object)namedPortal.getObject(_targetName); //retrieve the
target by its name from  the named portal

    if (_methodIndex == 0) {
     R=ReceiverObj2.run0(_methodArgs[0], _methodArgs[1]);
    }

    if (_methodIndex == 1) {
     R=ReceiverObj2.run1(_methodArgs[0]);
    }
   }
  Result=R;//assign the result as a return value

  } //end of doMethodCall
```

5.9 Summary

We aimed in this chapter to design a component framework for the RTSJ. This framework requires a set of software patterns that can be integrated to build predictable and reusable software components based on the RTSJ specification. To derive these design patterns, we proposed a procedure for dividing the design process into a set of design views; Business view, Thread mangaemnet view, Communication view, Memory Management view, etc. We studied the features of each view from RTSJ perspective, in order to derive the required patterns for each phase. This procedure resulted in a set of design patterns, the cornerstone of these patterns for is the Forked Memory design pattern, which is an RTSJ compatible component memory model that enforces the constraints of the RTSJ memory model, and provides easy-to-configure and use elements. We associated with the Forked Memory pattern a set of patterns that integrate with it, e.g. a life manager sub-component patterns to manage the lifetime of the scoped memory areas that constitute the proposed component model.

In order to have predictable component model, the patterns used to build the memory model have been designed with a consideration of reusing objects to avoid the waste of systems resources, where patterns for the reusability, e.g. object pools, and thread pools, are provided as sub-components that integrate with the component model. Moreover, we presented execution patterns for running reusable real-time tasks on the proposed component memory model.

Regarding the Communication view, we presented in this chapter the patterns that can be used for inner and inter communication, and we shows an example at the end of this chapter that apply the memory model and some of the associated patterns to build a method invoker that enables inter communication between two components residing in the same container.

We postponed the discussion of the local/remote communication to the next chapter; as it is dominant in our real-time middleware model, we present reconfigurable low level communication component that can be used either as a sub-component of other components, or as a separate component. This component integrates a set of communication patterns that support both synchronous and asynchronous communication, in order to be useful in various applications.

Chapter 6

Real-time Reconfigurable Communication Server Component Based on the RTSJ

In the previous chapter, we proposed a component framework to for developing our proposed middleware. In this framework, we showed the possible models of communication; inner-communication, inter-communication, local-communication, and remote communication. We showed that the models for both inner-communication and inter-communication are very similar, and presented the models of using them in the RTSJ. Also, we showed that the models for local communication and remote communication are very similar, but we have not presented models for implementing them in the RTSJ. As the remote communication models are essential part in any communication middleware, and as there are are various remote communication models; e.g. synchronous, asynchronous, our aim in this chapter is todesign an RTSJ based component for remote communication, using the framework presentd in Chapter 5, where this component can be configured to support different communication models as required.

As presented earlier in this thesis, the Real-time Specification for Java provides predictable memory and scheduling models for developing real-time systems using the Java language. However, it is does not provide communication mechanisms suitable for distributed real-time systems. So, the design of proposed component model aims to integrate the framework, which was presented in the previous chapter, and the other new features added by the RTSJ, with the available network packages provided in Java language, in order to support different models of communications used by many real time middleware solutions.

In our design, we assume that this proposed communication middleware component can be used either as a pluggable sub-component, which offers communication services to the functional component in order to support the communication facilities within the component model, or it can be built as a separate component, i.e. using the principles proposed in the framework, as presented in the previous chapter, where this component can be linked with other components to build the required model.

As presented in chapter 2, distributed systems have many different paradigms; however, the most commonly used is the client-server paradigm in which the communication is usually done concurrently among multiple clients and server(s) in order to achieve high performance and throughput. To achieve this high and efficient concurrency requirement, several methods and techniques have been provided. In order to build our own component, we need to study these different approaches and models; so, in this chapter, we provide an overview of the different I/O strategies used in distributed systems, either for issuing multiple concurrent I/O operations, or for concurrently handling clients' requests. Then, the several models for integrating different combinations of these strategies are overviewed with the support of the currently available underlying environment and operating system. Then, the Java packages' current support for networking communication models is presented.

We also present a review of the basic structure of three software patterns commonly used by servers, followed by a discussion of their pros and cons. Finally, we present our proposed model of a server component based on the RTSJ, which can be configured for reuse in developing real-time middleware solutions. In our presentation of the design of the component model, we will present its design as a sub-component, and how this design can be modified to make it a separate component using the framework presented in the previous chapter. At the end of the chapter, we provide an example that illustrates how the developed communicator component can be integrated with the patterns developed in the last chapter, for building a simple client-server application for exchanging packets of bytes.

6.1 Basic Strategies for Network I/O in Distributed Systems

Achieving efficient communication over networks has been a target for software developers and researchers to enhance the development of efficient distributed systems. Hence, a lot of research has been made toward optimizing these I/O operations; this resulted in the development of many strategies that are available for use in developing distributed systems. These strategies have been classified into two main groups as follows:

6.1.1 Strategies for issuing multiple I/O operations

Processing communication operations, i.e. sending or receiving data, over a network is much slower than local calls. So, the long time taken to make a communication operation does not just affect the performance of the calling thread, but also, it affects other threads that might block waiting for some data of the calling thread. This problem is very serious in the case of real-time systems, as it leads to the priority inversion problem when a thread of high priority has to wait for some other thread of lower priority that is blocked during making a call operation over the network. Moreover, the calling thread becomes inhibited from doing any other concurrent operations, during the execution of the remote call. To provide a solution for these problems, several strategies have been introduced to enhance the ability of a calling thread to make a communication operation over the network, the differences among these strategies comes from the fact that the communication operations are executed into two layers of the operating system, the user space, and the kernel space, see Figure 6-1. In the kernel space, the device drivers are responsible of executing the actual communication operation by sending/receiving the data associated with the communication operations from/to its associated buffers over the network stream; whereas the user space, holds the user buffer used by the calling thread in the user' program. Different scenarios can be used by the calling thread to transfer the data of the communication operation call from/to the user space to/from the kernel space. These scenarios lead to the following basic communication strategies:

A- Use blocking/synchronous calls from several threads.

This strategy adopts the blocking/synchronous method of communication. The processing of a synchronous call goes through the following steps (see Figure 6-2):

1- At a certain moment $T_{0,}$ the calling thread, which is running in the user space, issues an I/O operation, e.g. read().

2- Calling the synchronous I/O operation, causes the thread to switch to the kernel space in order to run the I/O synchronous operation, e.g. reading/writing the data from/to the device buffers in the kernel in order to transfer the data into/from the user buffer in the user space.

3- The thread checks the device buffers, if the device buffers are ready to make the I/O operation, the thread makes the transfer process and immediately returns to the user space, otherwise;

4- If the device buffers are not ready at T_1 to run the operation; e.g. they have no data in the case of the read () operation, the thread blocks are waiting for the buffers to become available.

5- Once the data arrives at T_2, and after the device buffers become available, the thread is activated to transfer the data from/to the device buffers into/from the user buffer.

6- Once the byte transfer operation finishes, at T_3, the call finishes and the thread becomes ready to execute other calls.

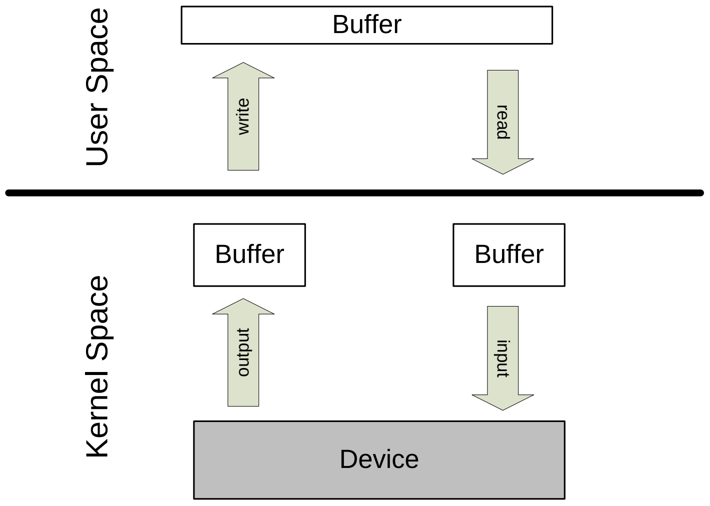


Figure 6-1 Buffers in the User Space and Kernel Space

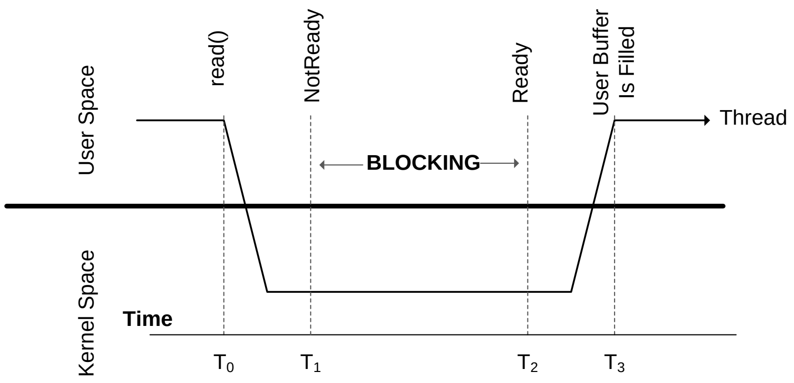


Figure 6-2 Execution of Synchronous I/O Operation

According to the scenario explained above, it is clear that the synchronous network I/O operations requires one thread for each concurrent network I/O operation and this in turn has the following defects:

1- Each thread requires its own execution stack; hence the memory size required will proportionally increase with the increase in the number of threads running concurrent I/O operations.

2- Increasing the number of concurrent threads running in the system will result in a high rate of memory references due to the increase in the number of context switches among threads of execution.

3- In many operating systems, the kernel has a limited number of data structures to schedule the concurrent running threads; e.g. file descriptors in Linux. So, using a single thread for each I/O operation will limit the number of the concurrent I/O operations to the maximum limit provided by the operating system.

So, this strategy does not facilitate the ability of making concurrent calls from a single thread, hence, to achieve the required concurrency, multiple threads have to be used; where each thread makes a single synchronous I/O operation call and waits for its completion.

B- Use non-blocking calls from a single/multiple thread(s)

This strategy uses the non-blocking techniques of communication where the calling thread can initiate the I/O operation on a non blocking socket in order to be able to do other tasks without waiting for the I/O operation to finish. The steps of executing a non-blocking call are as follows, see Figure 6-3.

1- Initially, at the time T_0, the calling thread, which is running in the user space, issues an I/O operation, e.g. read().

2- Once the thread calls the non-blocking I/O operation, the thread switches to the kernel space in order to see if the device buffers are available.

3- The thread checks the device buffers, if the device buffers are ready to make the I/O operation, the thread makes the transfer process and immediately returns to the user space, otherwise;

4- If the thread found the device buffers, at time T_1, are not available, the thread returns back to the user space

5- Once the thread returns back to the user space at T_2, it becomes free to make any other calls either I/O calls or other local calls.

6- Later, at a certain moment T_3, device buffers become available.

7- Hence, at any moment T_4, where $T_4 > T_2$, the thread tries to execute the same I/O operation, if the buffers are still not ready, i.e. $T_4 < T_3$, it again goes to the user space to do other operations; otherwise, if the device buffers are ready, i.e. $T_4 > T_3$, it transfers the data from/to the device buffers to/from the use space buffer.

8- Finally, when the I/O operation completes at T_5, the thread returns to the user space to continue execution.

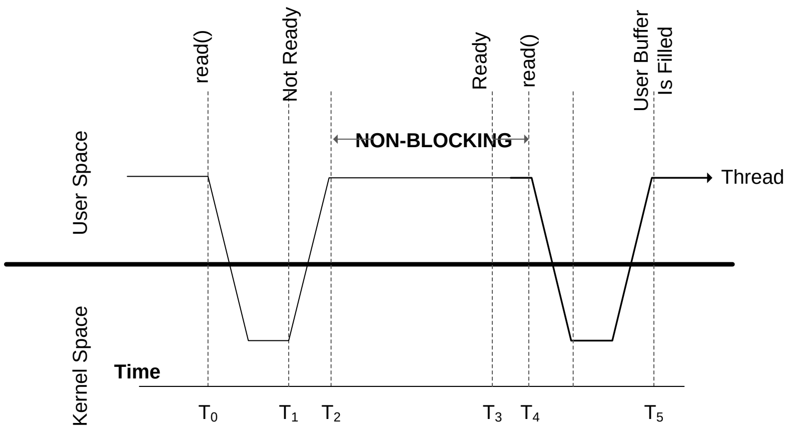


Figure 6-3 Execution of a non-blocking I/O Operation

The main benefit of using this strategy over blocking calls is that as the calling thread does not block waiting for the termination of the operation; hence, it can do other operations, either I/O operations or local operations, at the same time, which enhance the throughput and efficiency of the system. However, as explained in the above steps, as the calling thread won't block waiting for the completion of the slow I/O operation, a mechanism is required for declaring the readiness of the device buffers to complete the execution the I/O operation called by one of the non-blocking threads, this operation is well known as I/O multiplexing. Modern operating systems provide several mechanisms to support the I/O multiplexing, as will be explained later in this chapter.

C- Use Asynchronous calls from a single/multiple thread(s).

This strategy uses asynchronous methods to make the I/O operations calls. The steps of executing a non-blocking call are as follows (see Figure 6-4):

1- Initially, at the time T_0, the calling thread, which is running in the user space, issues an asynchronous I/O operation, e.g. `async_read()`.

2- The thread switches to the kernel space to enqueue its required I/O operation in the kernel's asynchronous I/O system, associated with a reference to the user space buffer if required and it returns back to the user space; this may even happen before the device buffers become ready at T_1.

3- Once the thread returns to the user space, at T_2, it runs asynchronously independent from the I/O operation.

4- The kernel's asynchronous I/O system monitors the device buffers required for the enqueued I/O operation, and once, the device buffers required for the I/O are ready, at T_3, the operating system, transfering the data from/to the device driver buffers to/from the user space buffer.

5- Finally, when the I/O completes, the operating system *asynchronously* interrupts the execution of the calling thread in the user space, at T_5, to notify it that its required I/O operation which has been completed in order to take any required action and it calls any call-back method registered with the enqueued I/O call.

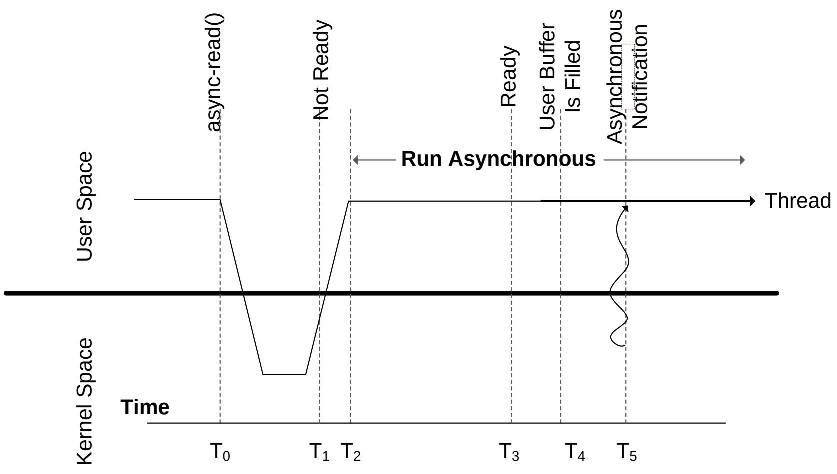


Figure 6-4 Execution of Asynchronous Calls

6.1.2 Strategies for Handling Clients' Requests

As it is normal for a server within a distributed systems to serve multiple concurrent requests from different clients concurrently, several approaches for providing handlers for these requests have been used:

One process for each client; this is the classic Unix approach, in this approach for each new client request the server *forks* a new process to handle it using the *fork()* command.

One Operating System thread for each client; this is the approach used by Java programs that run on a JVM that maps each Java thread into a *native thread*, i.e. it uses 1:1 mapping from user space threads to kernel threads. In this approach, the server creates a separate Java thread, which is a kernel thread, to handle the client request.

One Operating System thread for each active client; the thread pool approach is a very common example of this strategy. In the thread pool approach; the server has a pool of a predefined number of created and initiated worker threads waiting. With each new client request, the server retrieves one thread of this pool to handle the client request and once the thread finish handling the client request, it does not terminate, but it returns to the thread pool, to be ready for handling other clients' requests in the future. The main advantage of this strategy is that it limits the number of threads created in the system.

One Operating System thread handles multiple clients; the following three common approaches are good examples of using this strategy:

- ***Java with green thread;*** this approach is used in JVMs that uses n:1 mapping from user space threads to kernel space threads, i.e. all the Java threads are green threads that are created within the JVM process, where the JVM process itself is running in a single thread in the kernel level. Hence, all the green threads that are created by Java to handle the clients' requests will be processed by a single kernel thread.

- ***State machines;*** this approach assumes using a single thread for processing all clients' connections where the communication is using an event-driven mechanism. In this approach the same handling thread is called each time an event occurs on one of the connections in order to handle it. However, the thread cannot store in its own stack the individual processing state of each connection. In order to overcome this problem, this strategy assumes using a finite state machine attached with each connection, where the occurrences of the events on each individual connection drive the states' transitions of the state machine attached to this connection.

Examples of using this strategy is found in (JSpasm Open Source ; LimeWire Web Site), where the concept of finite state machines have been integrated with the event driven programming model provided by Java NIO to implement this strategy.

- ***Continuation.*** The processing of individual client requests requires independent paths of execution and it requires at the same time that each of those paths to provide the ability of saving the state of its associated connection along its path. This is exactly what the continuation techniques is assumed to do, as the continuation technique refers to the ability of saving the current execution state of a running execution unit, with the ability suspending and later resuming its execution. Hence, the continuation technique has been adopted by many researchers and developers for providing the server's request handling mechanism by integrating it with the Java NIO event facilities to provide an event driven programming model that use a single thread. This thread uses the continuation mechanism to save the state of the currently executing connection when another event occurs on another connections, and restoring this state back again when an event related to this connection occurs.

The difficulty of using the continuation mechanism is that it requires special support from the underlying programming language environment, for creating and managing independent execution paths/units within each individual thread. This support is not provided in many programming languages. A lot of research has been made to support continuations in the Java programming language, where several methods of implementations have been used; these implementations methods include (Jose A. Ortega-Ruiz, Curdt1 et al.):

1. Modifying the JVM
2. Hook into existing JVM though the JIT or JVMDI
3. Rewrite the byte code level
4. Rewrite the source code level

6.1.3 Models of Network Communications Processing

The previous two sections showed that there are various strategies that can be used for both multiple I/O operations and handling requests. This variation results in a set of different models for network communication processing. The most widely used among those models are (Kegel D. 2006):

A- Serve Many Clients with each Thread and use Non-Blocking I/O

In this model the kernel should provide the support of providing the readiness status of the I/O event through operating system interfaces. These operating system interfaces can be categorized into two sub-models, where each interface can support either one or both of these two submodels, these two submodels are:

- **Level Triggered Readiness Notification**; each time the calling thread requests the status of the I/O operation, the kernel notifies it if the I/O operation is ready for completion or not, whether the thread already has been told this in a previous call or not, i.e. the notification of the readiness status of a certain event is made as long as the I/O operation that caused this event has not been processed yet. Examples of operating systems interfaces supporting this sub-model are;

1. POSIX commands *select()*, and *poll()*, and
2. Solaris's command */dev/poll*, and
3. The level triggered version of FreeBSD's command *kqueue*.
4. The level triggered mode of the *epoll()* provided by the Linux kernel *v2.6+*.

- **Edge Triggered Readiness Notification**; the kernel notifies only the requesting thread when the first time the I/O operations becomes ready, and any next requests from the thread will not be notified even if the I/O operation has not been processed yet, as the kernel will assume that the thread already knows that the I/O operation is ready.

Examples of operating systems interfaces supporting this sub-model include:

1. The edge triggered version of FreeBSD's command *kqueue()*.
2. Real-time Signals by the Linux kernel *v2.4+*.
3. The edge triggered mode of the *epoll()* provided by the Linux kernel *v2.6+*.

B- Serve Many Clients with each Server Thread and use Async. I/O

In this model the kernel has to have the support of asynchronous I/O over the sockets in order to asynchronously notify the thread of the completion of the I/O operation. Unfortunately few operating systems provide asynchronous I/O over sockets. The Windows Operating System is one of those, as it provides it through the use of the *I/O completion ports*, whereas the Linux operating system provides support for asynchronous I/O over files only but not over sockets.

C- Serve One Client with each Server Thread and Use Blocking I/O

This method is supported by many operating systems as it is based on the classical blocking I/O POSIX operations; *read()* and *write()*. However, to ensure that only a single kernel thread will handle only the request of a single client, this method requires the operating system threading library to support the 1:1 thread mapping from user space to kernel space. Examples of such threading libraries are the IBM's NGPT and the NPTL POSIX-Compliant threading libraries for Linux which are both using 1:1 thread mapping method.

D- Kernel Built In Server

The final model assumes that instead of building the server in user space, it should be built in the kernel itself in order to minimize the number of hooks required to enhance the server performance. An example of this type of server is the *khttpd* web server for Linux; it is built as a kernel module to act as *http server* for handling static web pages in Linux.

6.2 Low Level NW Communication Support in Java

The Java language has been used to build a great number of applications, components, and packages that run on distributed systems as the language offers good and easy to use networking packages which hide many of the complications of communications over networks. The Java language supports networking communication mainly through the following three main packages:

- **Networking Package;** this package is encapsulated in the `java.net` package and it was introduced in JDK 1.0 but additions, enhancements, and modifications have been made to it later; especially in J2SE 1.4 to integrate its classes with the NIO packages.
- **Standard I/O Package;** This package is encapsulated in the `java.io` package and it was introduced in JDK 1.0 to support very basic blocking communication mechanisms.
- **NIO Package;** This package is encapsulated in the `java.nio` package. This package was a result of the work done in JSR51 (JSR-051) to complement and enhance the standard I/O package and it was first introduced in J2SE 1.4. It has got this name as it provides New I/O package, or as it supports Non-Blocking I/O.

These three packages provide support for general I/O operations and not just networking operations; e.g. it supports file manipulation operations. However, we will discuss here only the very basic networking support offered by all of these packages and see how much support they do offer to satisfy the requirements for building real-time systems.

6.2.1 Networking Package

This package provides a set of classes dedicated for networking in Java, the most important classes in this package are the following:

- `java.net.ServerSocket`; this class represents the server side end point at one side of a two way communication link between two programs running on a network. This class mainly offers the *accept* () operation required by the server side of the connection to monitor the arrival of incoming connection requests from the clients, and it implicitly creates sockets for each accepted connection at the server side a new socket of the type `java.net.Socket` class that communicate with the client's socket over the network stream.
- `java.net.Socket`; this class does not just offer the *connect()* operation required by the client side of the connection in order to initiate a connection with the server, but it also provides operations to access both the input and output stream of the connection both at the server side and at the client side. The `java.net.InetSocketAddress` is a subclass of this type that implements IP socket addresses.
- `java.net.DatagramSocket`; this class represents a socket that sends and receive - datagram packets of the type `java.net.DatagramPacket` over the network. The

`java.net.MulticastSocket` is a subclass of this class that enable sending and receiving IP multicast packages over the network.

- `java.net.SocketAddress`; this abstract class represents a socket address with no specific protocol. Its sub class `Java.net.InetAddress` implements an IP socket address of an endpoint which consists of IP address and a port number.

6.2.2 Standard I/O Packages

This package has been built assuming the representation of the I/O operations in a form of the Reading/Writing Stream model. In the case of networking, the stream represents a sequence of transmitted bytes over the network, see Figure 6-5. Hence, the I/O operations have been represented according to this model by the following classes:

- `java.io.InputStream`: This is an abstract class, and it is the super class of all classes representing an input stream of bytes, i.e. classes extending this class will represent a sequence of bytes coming as an input to the system, where the stream can be network stream, file stream, etc.

The basic operations offered by subclasses implementing this class are mainly *read* operations that enable reading either the next single byte of the stream, using *read()* method, or the next group of n bytes from the incoming stream, using *read(byte[] b)*. These read operations are blocking operations, i.e. the calling thread will block waiting for a result if there is no data available on the input stream.

- `java.io.OutputStream`: This is another abstract class, and it is the super class of all classes representing an output stream of bytes, i.e. classes extending this class will send a sequence of bytes as an output to the target stream.

The basic operations offered by this class are *write* operations to write either a single byte or group of bytes to the end of the output stream.

Limitations of the Standard I/O package

The limitations of the standard I/O package come from the fact that the standard I/O package is mainly dependent on using blocking operations for accessing the networking stream. This can be clarified by the diagram shown in Figure 6-6 through the following steps:

1. First the server waits for the call request using the blocking method *accept*().
2. Once the request arrives, the server starts to read the client request using another blocking method *read*().

Finally, to send the result back to the client, the server calls a third blocking method *write* ().

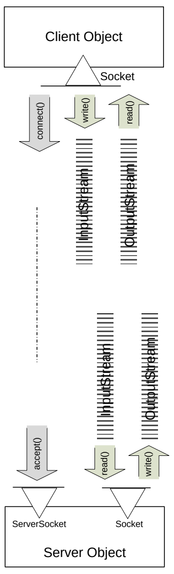


Figure 6-5 Integration of Sockets with the Stream Model in Java

When running the server as a single threaded program, the blocking nature of the calls will inhibit the calling thread from doing any useful work during this time, and this will affect the performance of the server; this problem is even more serious if the server is required to run within a real-time environment, as it may lead to priority inversion if other threads with higher priority have to wait for this server thread to continue execution.

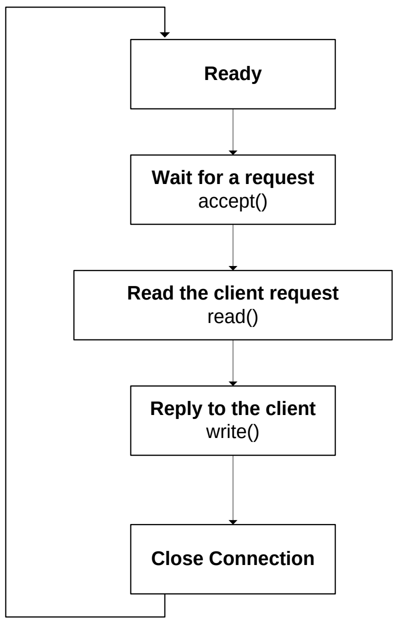


Figure 6-6 Basic Server Handling of Client Requests using Standard I/O Package

The usual solution to overcome this problem is to use multi threading, where the server thread becomes responsible for monitoring the requests coming from clients, and initiate a dedicated thread, or reusing a thread from a thread pool, for handling the request and replying to the client. Unfortunately, this solution is not scalable and adds a lot of overhead to the system, due to the problems arising from using multiple concurrent threads concurrently, which is not just because it is always limited by the physical resources, but also due to the fact that increasing the number of concurrent threads affect the response time and the predictability of the execution of the server, and all other threads within the system.

6.2.3 Java NIO Package

The Java NIO package is not a replacement of the Standard I/O package, but it complements and extends its functionalities. The aim of introducing it is to provide an enhanced access and manipulation of the I/O operations supported by many new operating systems, in order to overcome the limitation imposed by using the standard I/O package in Java, as mentioned in the previous section. 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 (The JSR-203), but we will not consider these additions here, as they are going to be part of JDK-7 which is not supported by any JVM that supports RTSJ.

The main force behind the Java NIO is the inefficiency of the blocking model of the standard package, so Java NIO adopted the non-blocking Reactor pattern (D. C. Schmidt 1995), as a more efficient model for communication. It is required that a Java server object running in a distributed system has to be able to handle multiple concurrent client requests at the same time efficiently. The Reactor pattern is assumed to provide a higher efficiency than that which is provided by the blocking methods found in the Standard package. The idea of the Reactor pattern is to consider the clients' requests as events occurring on the networking stream that it, the Reactor, has to react to by multiplexing theses events, and dispatching a dedicated service provider of each event in order to reply to as many events as possible concurrently.

To provide the Reactor pattern, Java NIO package has provided a set of important features. The most important features offered in the Java NIO package, from the networking point of view, are the following.

A- The NIO Byte Buffers

These are buffers that hold the data to be transferred by channels to/from the networking stream. They can be created to buffer raw bytes using the class `java.nio.ByteBuffer`, or they can be created to buffer primitive types, where each primitive type, except the `boolean` data type, has its own subclass e.g. `java.nio.IntBuffer`**.** The interesting thing, about NIO buffers, that they can be allocated either *direct* using the factory method *ByteBuffer.allocateDirect()* or *indirect* buffer using normal constructors. The difference between the two types as was mentioned in the Java SE API Javadocs (Java SE API) is as follows:

"A byte buffer is either direct or non-direct. Given a direct byte buffer, the Java virtual machine will make a best effort to perform native I/O operations directly upon it. That is, it will attempt to avoid copying the buffer's content to (or from) an intermediate buffer before (or after) each invocation of one of the underlying operating system's native I/O operations. "

This can be clarified more by the diagram shown in Figure 6-7; as the diagram shows in both cases the direct allocation and the non-direct allocation, the byte buffer object itself is created in the heap; the difference will be mainly in the place of the allocation of the inner buffer of the byte buffer object that will hold the bytes of the I/O operation. As shown in the diagram, in the case of the direct allocation, this inner buffer will be allocated outside of the heap, whereas in the case of the direct allocation, it will be allocated within the heap. Hence, in the case of the direct buffers, the garbage collector will have no interference with the inner buffer, and hence the allocated space cannot be moved by the garbage collector.

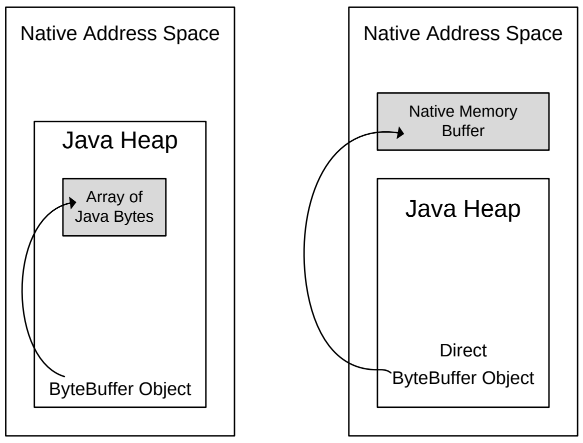


Figure 6-7 Direct and Non-Direct Byte Buffer Allocation

B- The Selectable Channels

A channel is a new abstraction of representing the endpoint of communication link on a networking stream. This does not mean that it replaces completely the older abstraction, i.e. the socket, but it complements and extends it. In Java NIO each networking channel has still to be attached to a socket. But, it expands the range of operations that can be offered by the endpoint over the networking stream.

Channels can be seen as tubes that are responsible for transferring data efficiently to/from the byte buffers, as mentioned next, to the networking stream. There are three main types of networking channels offered by the packages, and all of them are inheriting their common properties from the `java.nio.channels.SelectableChannel` class:

Server Socket Channel; this type is created from the class `java.nio.channels.ServerSocketChannel`. It is responsible only for listening for incoming connection requests, and when it receives a request it creates another dedicated Socket channel object to handle this request.

Socket Channels; this type is created from the class `java.nio.channels.SocketChannel` and it is responsible for reading and writing operations on the networking stream socket object, created from the class `java.net.Socket`**,** attached to it using stream oriented connections.

Datagram Channels; this type is created from the class `java.nio.channels.DatagramChannel`. Like the Socket Channels, objects of this type are responsible for reading and writing through the network stream. However, each object created of this type is attached to an instance of `java.net.DatagramSocket` through which it accesses the stream using the datagram protocol.

C- The Selector

A selector object is created from one of the subclasses inherited from the `java.nio.channels.Selector` class. The selector plays the central rule in mapping the Reactor pattern to the Java NIO as it, the selector, is responsible for monitoring the I/O events occurring on the stream, see Figure 6-8. Once an event occurs on a registered channel this event is captured by the selector in order to be dispatched to the corresponding selectable handler to handle it.

A selectable channel is registered with a certain Selector via the channel's *SelectableChannel.register()* method. The *SelectableChannel.register()* method takes two parameters:

1- The Selector with which the Selectable Channel is to be registered.
2- The set of the events this Selectable Channel is interested and wants the Selector to monitor them. These operations include:

- `SelectionKey.OP_ACCEPT`: The SelectableChannel is of a Server Socket Channel type and it is ready to receive a request from a client to establish a new connection.
- `SelectionKey.OP_READ`: The selectable channel is ready to read data when it is available on the input stream.
- `SelectionKey.OP_WRITE`: The selectable channel is ready to write to the output stream when it is ready.
- `SelectionKey.OP_CONNECT`: The selectable channel is ready to *complete* its connection sequence.

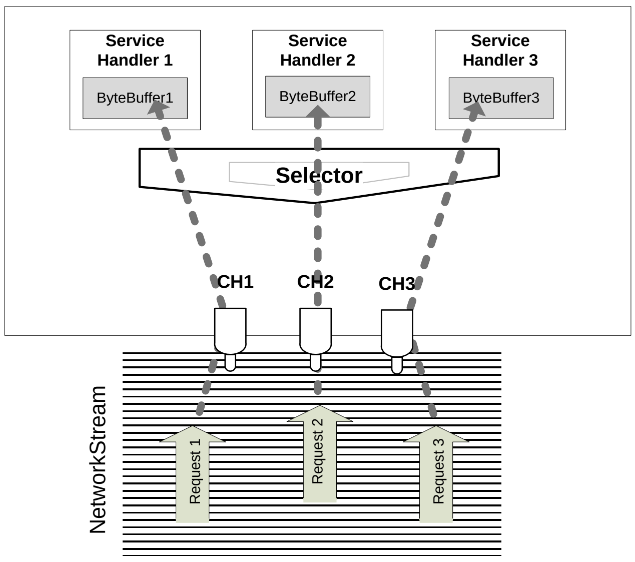


Figure 6-8 I/O Multiplexing using Selectors in Java NIO

Each registration process of a Selectable channel with a Selector creates a Selection Key Object created from the class `java.nio.channels.SelectionKey`. The Selector maintains three sets of selection keys as follows:

1- **The Key Set;** this set maintains all the selection keys of all selectable channels currently registered with this selector.

2- **The Selected Key Set;** this is a subset of the Selection Key Set. It maintains a list of all keys of selectable channels that are detected to receive at least one of the occurred events on the stream.

3- **The Cancelled Key Set;** this is another subset of the Key Set. It maintains all keys of selectable channels that have been cancelled, i.e. the selector is not monitoring interested operations defined for them, and their channels have not been deregistered yet.

The selection operation, i.e. the operation of monitoring an event and dispatching it to its corresponding handler, is done by calling one of the *select()* methods of the Selector Object. There are three versions of the selection operation:

- `Selector.select ()`: This method performs a blocking selection operation waiting for the occurrence of any of the defined set of events. However, it can be interrupted by calling the *wakup()* method or if the thread calling it is interrupted.
- `Selector.select (long timeout)`: This method is similar to the `select()` method with the added ability to return after the expiration of the provided timeout. However according to the Java NIO specification, this method does not offer real-time guarantees as it schedules the timeout similarly to calling `Object.wait (long)` method.
- `Selector.selectNow()`: This method performs a *Non-blocking* selection operation, i.e. it returns immediately if there are no available events on the stream, of those events which the selector is registered to monitor them.

6.3 Server Model 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. We will review three server models: multithreaded synchronous, reactive synchronous and proactive asynchronous. In this review we will present the basic architecture of each model followed by a discussion of its pros and cons.

6.3.1 Multithreaded Synchronous Server

In this pattern, 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 (M. Voelter 2004), which is an example of this category, has the following main elements:

- **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.
- **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 (D. C. Schmidt 1995; D. C. Schmidt et al Aug. 2000):

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- The 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 a very sensitive problem for many real time systems with very limited resources.

Some variations of these 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 thread pool 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* pattern (D. C. Schmidt et al Aug. 2000), 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.

6.3.2 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 (D. C. Schmidt 1995). 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)*;*** defines 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 acknowledgement of blocking possibility, hence the operation can be repeated later to avoid blocking the server thread.

As the principle 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 (D. C. Schmidt 1995; Pyarali 1999):

1- **Portability of the Design among Many Operating Systems**; as it does not need multi-threading, it can be built on any operating system.
2- **Low Concurrency Overhead**; as there will be no context switching, nor synchronization, as it is using single threaded model.
3- **Modularity**; as it decouples the application logic from the dispatching mechanisms.

Although the Reactor pattern has many good features, the reactive server pattern has a set of drawbacks including (Pyarali 1999):

1- **Program Complexity;** the server logic can be very complicated to avoid blocking the server during handling client requests.
2- **Less efficiency for multithreaded systems;** it adopts a single threaded model; hence, it cannot utilize the hardware parallelism effectively. Hence, it is less efficient for servers on multicore hardware.
3- **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.

6.3.3 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 (Pyarali 1999) is a non-blocking asynchronous pattern. The key elements of this pattern, as shown in Figure 6-9 are:

1- ***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.

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

Asynchronous Operations*;* these are the operations to be executed by the operating system on behalf of the server application.

Asynchronous Operation Processor; this is the operating system implementation responsible for executing the asynchronous operation, and notifying the completion dispatcher when finished.

Completion Dispatcher; this is responsible for 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.

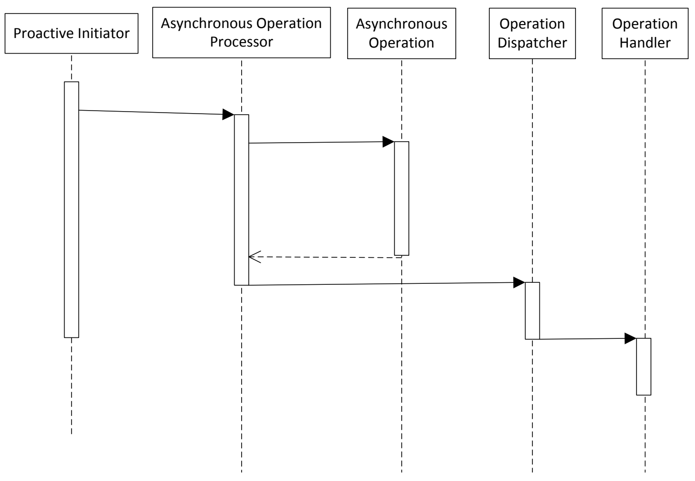


Figure 6-9 Proactor Design Pattern

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

1- **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.

2- **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.

3- **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.

4- **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.

5- **Higher performance;** the call-back notification mechanism, used by the asynchronous operation processor to notify the completion of asynchronous operations, enhancing 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.

6- **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:

1- **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.

2- **Lack of the order of execution;** 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.

6.4 Design of a Configurable Communicator Component

Due to the variations in the communicating requirements of real-time middleware solutions, there is a need to provide software reconfigurable components that enable the real-time middleware developers to configure and use them according to the functionalities and requirements proposed in their target middleware solutions. As there are several paradigms of communications, it is impossible to combine them all into a single component. In our work, we are considering only the Client-Server Communication Paradigm.

In our design, we assume that this proposed communication middleware component can be either used as a pluggable sub-component that offers communication services to the functional component to support the communication facilities within the component model or it can be built as a separate component, i.e. using the principles of the framework as presented in the previous chapter, where this component can be linked with other components to build the required model.

In this section, we present our proposed design of a general model for a real-time communication component that integrates the predictable memory and scheduling model provided by RTSJ with the current networking and communication packages provided by Java. This real-time communication component is assumed to provide a set of reconfigurable properties from which the programmer can configure several properties of the component including its scheduling and concurrency policy, etc, that are required for the design of his own real-time middleware. The design of this real-time component will be using the proposed RTSJ component model presented in the previous chapter.

In the following, we will first present the basic elements that form our component, then we will discuss how it can be built using the RTSJ and what are the reconfigurable properties of this component that define its behavior in order to adapt to the requirements of the environment in which it is going to be used in.

6.4.1 Internal Elements

In the design of our real-time communication component we assume that the component should provide flexibility in configuration so it can support as many as possible of the server strategies and concurrency models discussed in this chapter. 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 some other models). Hence, from our understanding of the Proactor pattern, we assume the communication component has the following elements, see Figure 6-10:

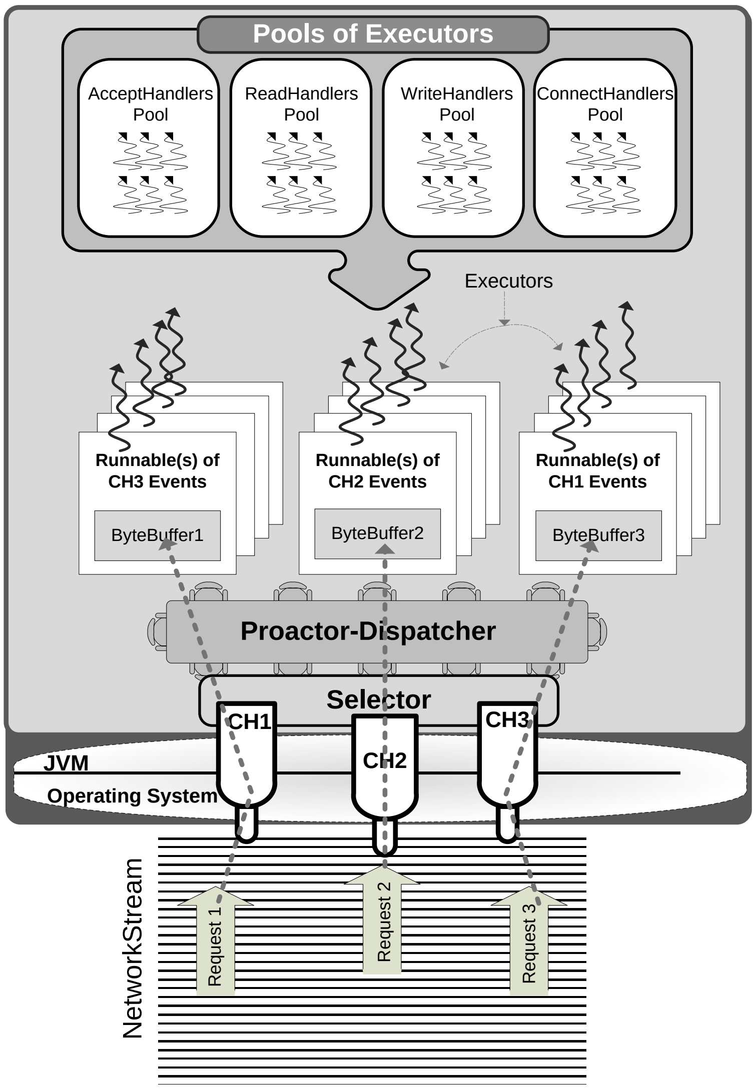


Figure 6-10 Elements of the Communicator Component

- ***Selectable Channels;*** these are the communication endpoints used by the component; these channels can be classified as server channels or client channels. 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 component is configured to be synchronous, or non-blocking, when the server is running as reactive or proactive.

- ***Selector;*** this element blocks waiting for the notifications coming from the JVM/OS, to indicate the occurrence of certain predefined events occurring at the channels, i.e. the readiness of the device buffers to complete the I/O operation, in order to notify the *Proactor Dispatcher* to react to these events.

- ***JVM/OS Operation Processor***; this is the interface provided by the JVM to forward processing of the asynchronous operation to the operating system. Again, this can be configured to use one of the commands offered by the operating system, e.g. select, poll, etc.
- ***Proactor Dispatcher;*** both of the *Proactor Initiator* and the *Completion Dispatcher* parts of the Proactor pattern are combined and integrated within this element. This element is responsible for initiating the communication event handling 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, the Proactor dispatcher registers with the operation a completion handler. So that, 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. create a new handler for every new client request or reuse one free handler from a handlers pool.
- ***Runnable(s) of the Event Handler(s);*** these are the set of operations that are executed by the completion handlers in order to react to the events occurring on the channels. These operations can be executed either synchronously or asynchronously according to the component synchrony policy. Basically, there are four events defined on the channels monitored by the server, i.e. connect, accept, read, and write; hence, for each event we can define a separate Runnable, i.e. ConnectorHandler's runnable, AcceptorHandler's runnable, ReaderHandler's runnable, and WriterHandler's runnable.
- ***Executors (Completion Handlers);*** these are the processing units that act as event handlers for the communication events and they are responsible for executing the logic defined by the user, i.e. they execute the operations defined in the *runnables* of the event handlers mentioned in the previous element.
- ***Executors' Pool(s);*** each one of these elements manages a reusable set of executors. Where each Pool corresponds to the executors of a single event, i.e. a Pool for ReadHandlers, another for WriteHandlers, etc. Each one of these pools can optionally have a fixed pre-configurable size in order to limit the concurrency of the component, where this size can be detrmined by analyzing the system in which this component is to be used, in order to find the maximum number of concurrent executors used within the system, this is explained in more details for both the server side and the client side in Chapter 7, pages 277, 283 respectively.

6.4.2 Design and Configuration of the Component

In our design of the communication model, the communicator, we assume that it can be designed either as a sub component within our framework model, or it can be a separate component that is implemented using our proposed framework, which was presented in the last chapter. In the first case, i.e. the design as a sub-component, the communicator can be implemented as an object that can be optionally created within the container; so that, it offers its services to the inner components within this container, see Figure 6-11.

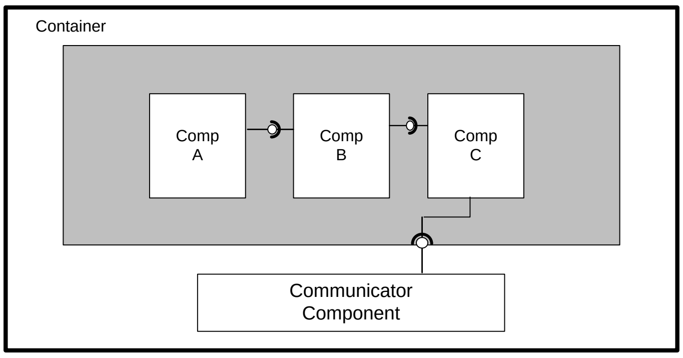


Figure 6-11 The Communicator as a sub-component

In the other case, i.e. the case of designing it as a separate component, the communicator component has to be designed using the framework model developed in the last chapter, where it is linked to the other component within the same container just as any component in the framework, i.e. By making the other components to implement the `IBindController` to access and use its interface, see Figure 6-12.

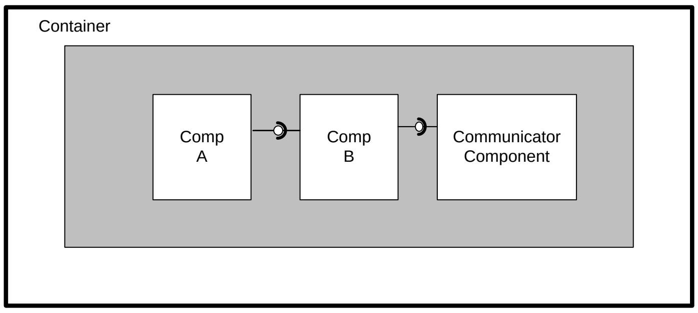


Figure 6-12 The Communicator as a separate Component

The functionality of the communicator component requires the running of a set of concurrent handlers/executors at the same time within its structure, where these handlers are used for the duration of handling the communication events and have to be recycled again. This functionality is a direct mapping to our component model, where the executors/handlers are the concurrent tasks that are running within the component boundaries, and each task has its own scoped memory stack, where all the executors/handlers are sharing the same CMA of that component, and they can use it for the inner communication. Also, these components area created in the container memory area by using the reusable schedulable object subcomponent, in order to be recycled for reuse after finishing the handling of the events, to limit the concurrency within the component. In addition to that the events handler/executor may need to run operations that create objects during their lifetime; so, in order to bound the memory usage within the component, these handlers have to create these temporary objects either:

- **As a reusable objects,** i.e. using the allocator. In this case the handler/executor can allocate the handlers/executors in the CMA/ContMA, depending on the operation, i.e. CMA is used when these objects are to be used only within this component, whereas ContMA is used for allocating objects that can be used either inside the component or outside the component with other components within the same container.
- **As temporary objects in scoped memory areas with short-lifetime,** so that they can be reclaimed after finishing the execution. In this case the SMA stack assigned for each executor/handler can be used to run these operations, or it may need to use separate scoped memory areas for sending messages to other objects or among themselves; in this case, these handlers/executors can use the scoped mmory life-managers to manage the life time of these scoped memory areas.

In both cases, the handlers can share the objects, either with themselves or with other threads from other object that share with this component its forked memory area. In this case, to enable the communication, the handlers use the multi-named object to save the shared object in the portal of the CMA or ContMA as required.

As it uses the forked memory model, these handlers can execute their logic for handling the communication events, using the executable runnable stack pattern defined for use with the forked memory patterm.

We can see from the above functionality that the communicator component is a direct example of using the memory model, which we presented in the previous chapter. This execution model can be defined as a dynamic forked execution pattern over the forked memory model. The dynamicity of this pattern is due to the dynamicity of the inner tasks/executors running within it, as they have a lifetime shorter than than component and they are reused frequently. This makes this forked execution pattern different from the staic pattern that results from using a fixed set of tasks, that are used for the lifetime of the component and they are never reused.

So, in the following section, where we develop the design of the component using the component model defined in our framework, where we show the design of the communicator as a sub-component; then, we show how the design can be adapted to build it as a separate component.

To be reconfigurable, the software component should provide assignable properties through its external interfaces that can be used by the developer, in order to configure it before initializing it. Internally, these configurable properties can either be used for initializing the member-variables of the inner objects constituting the component, or it selects certain behaviour of one of the inner elements within it. Also, the values assigned to these properties can be either from a predefined small set of predefined values, as in the case of the synchrony policy property, or it can be assigned arbitrarily from a wide range of values, as in the case of assigning a network address for the component. In this section, we present our design of the real-time communication component, where we will present the design of each class representing one of its inner elements within it. As we mentioned earlier, we will show the design of each element in two views; the first view for a sub-component design, and the other view for designing a separate component.

A- The Communicator as a Sub-Component (Proactor Dispatcher)

The Proactor dispatcher is the main central element of the component; hence, instead of modelling it as a separate entity, other than the entity representing the component itself, we will consider the Proactor Dispatcher as the main entity of the communicator component.

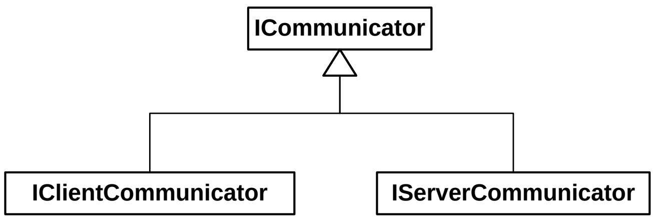


Figure 6-13 Interfaces of the Communicator Component

The communicator component can be used either at the server side, or at the client side. As there is a slight difference of the operations that can be made by the communicator at both sides; in our design of a class of the component we assume that the common properties and operations of the communicator are represented in an abstract class, the `Communicator` class, that implements the `ICommunicator` interface, which defines the common operations offered by this component; wheather it is configured as a client or as server. Two subclasses are extending this abstract class, one to represent the Proactor dispatcher as a client, the `ClientCommunicator` class and another one to represent it as a server, the `ServerCommunicator` class. These two classes implement the `ICommunicatorClient` and `ICommunicatorServer` which extends the `ICommunicator` interface to define the specific operations offered by these two components, see Figure 6-13. These classes and their properties and operations are shown in the class diagram in Figure 6-14.

This diagram shows the class diagram which is used to create the communicator as an object that works as a subcomponent within the container. The main elements of this class diagram are explained next:

Communicator Class

This is an abstract class, which represents the skeleton of the component, wheather it is employed to work as client component, or as a server component. In our model, the logic of the component is running within a real-time thread defined within this class by the `driverThread` reference, this thread is responsible for running the events monitoring task; e.g. event pooling loop, within the component. Moreover, the component can run either at the server side for accepting and managing requests or, at the client side, to make connections with servers and managing requests on them. The type of the component is identified internally using the `commType`, which is assigned the type of the component at the time of creation. In order to provide the component's functionalities in our model, this class defines the three groups of operations: (1) creation operations, (2) internal operations, and services operations, these are explained next:

The first group of the operations includes the static creational operational, and it is responsible for creating the object instance of the component, and it includes two main methods:

1- `createClientCommunicator(…);` this is a static method for creating instance(s) of the communicator component, to work at the client side, and it is used to initialize the component with the following properties:

`inetAddress;` this is used to specify certain I/P address (network address, port number) which represent the endpoint to which the component will connect, and make the requests for communication; if it is a client-side component, or, in the case of the server-side component, it will represent the endpoint on which the component will receive the requests.

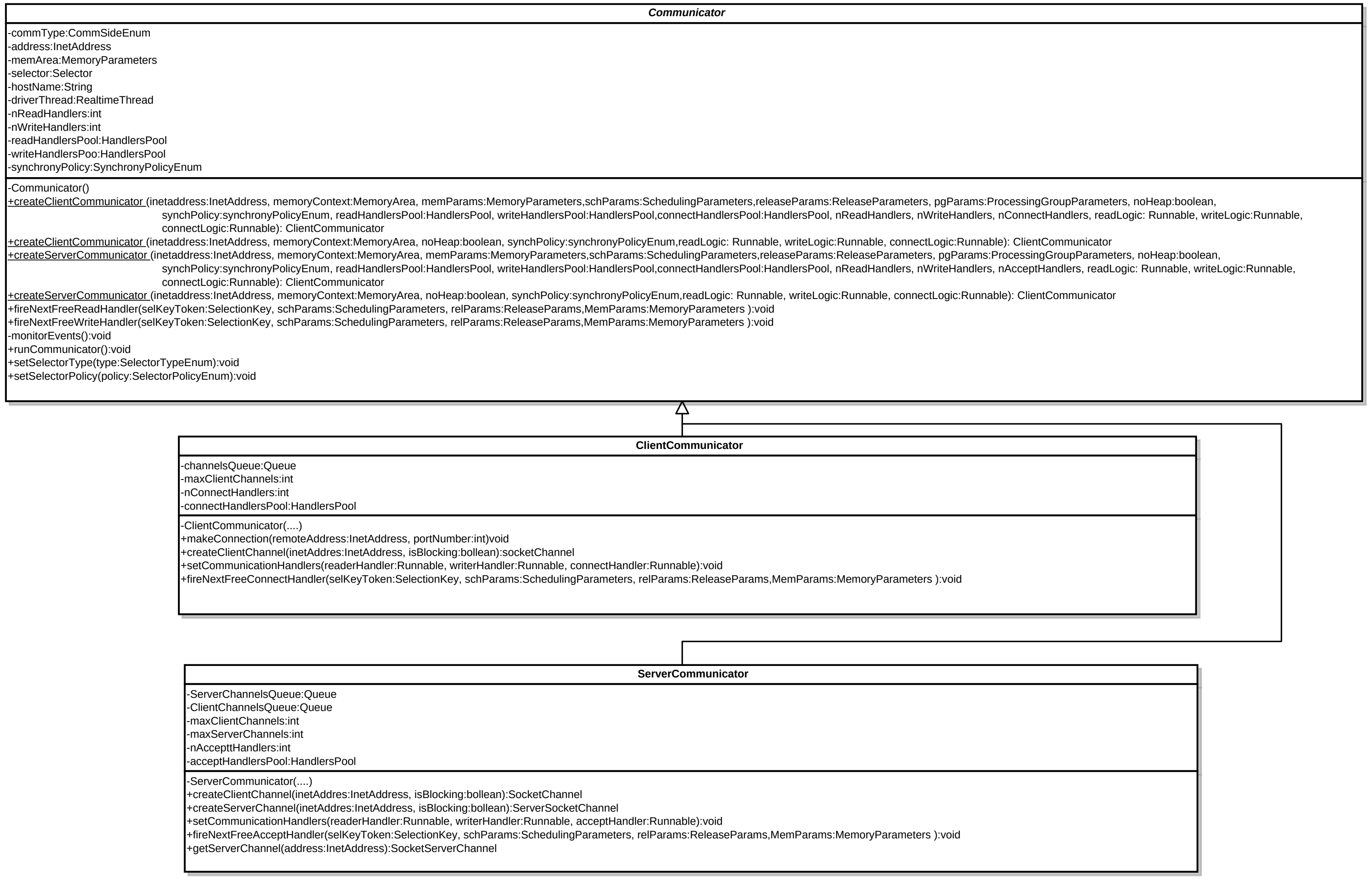

Communicator
-commType:CommSideEnum
-address:InetAddress
-memArea:MemoryParameters
-selector:Selector
-hostName:String
-driverThread:RealtimeThread
-nReadHandlers:int
-nWriteHandlers:int
-readHandlersPool:HandlersPool
-writeHandlersPoo:HandlersPool
-synchronyPolicy:SynchronyPolicyEnum
-Communicator()
+createClientCommunicator (inetaddress:InetAddress, memoryContext:MemoryArea, memParams:MemoryParameters,schParams:SchedulingParameters,releaseParams:ReleaseParameters, pgParams:ProcessingGroupParameters, noHeap:boolean, synchPolicy:synchronyPolicyEnum, readHandlersPool:HandlersPool, writeHandlersPool:HandlersPool,connectHandlersPool:HandlersPool, nReadHandlers, nWriteHandlers, nConnectHandlers, readLogic: Runnable, writeLogic:Runnable, connectLogic:Runnable): ClientCommunicator
+createClientCommunicator (inetaddress:InetAddress, memoryContext:MemoryArea, noHeap:boolean, synchPolicy:synchronyPolicyEnum,readLogic: Runnable, writeLogic:Runnable, connectLogic:Runnable): ClientCommunicator
+createServerCommunicator (inetaddress:InetAddress, memoryContext:MemoryArea, memParams:MemoryParameters,schParams:SchedulingParameters,releaseParams:ReleaseParameters, pgParams:ProcessingGroupParameters, noHeap:boolean, synchPolicy:synchronyPolicyEnum, readHandlersPool:HandlersPool, writeHandlersPool:HandlersPool,connectHandlersPool:HandlersPool, nReadHandlers, nWriteHandlers, nAcceptHandlers, readLogic: Runnable, writeLogic:Runnable, connectLogic:Runnable): ClientCommunicator
+createServerCommunicator (inetaddress:InetAddress, memoryContext:MemoryArea, noHeap:boolean, synchPolicy:synchronyPolicyEnum,readLogic: Runnable, writeLogic:Runnable, connectLogic:Runnable): ClientCommunicator
+fireNextFreeReadHandler(selKeyToken:SelectionKey, schParams:SchedulingParameters, relParams:ReleaseParams,MemParams:MemoryParameters):void
+fireNextFreeWriteHandler(selKeyToken:SelectionKey, schParams:SchedulingParameters, relParams:ReleaseParams,MemParams:MemoryParameters):void
-monitorEvents():void
+runCommunicator():void
+setSelectorType(type:SelectorTypeEnum):void
+setSelectorPolicy(policy:SelectorPolicyEnum):void
ClientCommunicator
-channelsQueue:Queue
-maxClientChannels:int
-nConnectHandlers:int
-connectHandlersPool:HandlersPool
-ClientCommunicator(....)
+makeConnection(remoteAddress:InetAddress, portNumber:int)void
+createClientChannel(inetAddres:InetAddress, isBlocking:bollean):socketChannel
+setCommunicationHandlers(readerHandler:Runnable, writerHandler:Runnable, connectHandler:Runnable):void
+fireNextFreeConnectHandler(selKeyToken:SelectionKey, schParams:SchedulingParameters, relParams:ReleaseParams,MemParams:MemoryParameters):void
ServerCommunicator
-ServerChannelsQueue:Queue
-ClientChannelsQueue:Queue
-maxClientChannels:int
-maxServerChannels:int
-nAccepttHandlers:int
-acceptHandlersPool:HandlersPool
-ServerCommunicator(....)
+createClientChannel(inetAddres:InetAddress, isBlocking:bollean):SocketChannel
+createServerChannel(inetAddres:InetAddress, isBlocking:bollean):ServerSocketChannel
+setCommunicationHandlers(readerHandler:Runnable, writerHandler:Runnable, acceptHandler:Runnable):void
+fireNextFreeAcceptHandler(selKeyToken:SelectionKey, schParams:SchedulingParameters, relParams:ReleaseParams,MemParams:MemoryParameters):void
+getServerChannel(address:InetAddress):SocketServerChannel

Figure 6-14 Communicator Class Diagram

- `memoryContext;` this defines one of the memory areas defined by the RTSJ memory areas 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; hoever, the assignment of a scoped memory area (`LTMemoryArea`, or `VTMemoryArea`) would be the best choice for servers with shorter lifetime duration.
- `memParams, schParamss, relParams, pgParams`; these parameters are used for creating and running the `driverThread`.
- `noHeap;` this parameter is used to indicate if the *driverThread* is specialized to be of the type `NoHeapRealtimeThread` or not.
- `synchronyPolicy`; this member of the class defines the synchrony policy of the server, i.e. it defines the behavior of the reacting to the communication events. It should have one of three values (1) *Procative*, (2) *Reactive*, (3) *Synchronous*, the details scenarios of execution of these policies will be presented later in this chapter in part B- of section 6.4.3.
- `readHandlersPool`, `writeHandlersPool`, `connectHandlersPool`; to specify the pools for all the read handlers, the write handlers and the connect handlers associated with this component.
- `nReadHandlers`, `nWriteHandlers`, `nConnectHandlers;` to specify the maximum number of the read, write and connect handlers used within this component respectively.
- `connectLogic`, `readLogic`, `writeLogic`; these parameters specify the runnable logic defined by the developer using this component for handling connect, read, and write events respectively**.**

`createServerCommunicator(…);` this is another static method that is used for creating instances of the communicator component to work at the server side. This method initializes the component with properties similar to those defined above for the client side, with the exception that the `inetAddress` in this method, defines the local network address that the component is required to monitor the events occurring on it. Moreover, instead of the `connectHandlersPool`, `connectLogic` and `nConnectHandlers`, the server communicator uses the `acceptHandlersPool`, `accepttLogic` and `nAcceptHandlers` respectively.

The second group of operations include the internal operations that are used internally by the component, these operations include:

- `fireNextFreeReadHandler(…)`,`fireNextFreeWriteHandler(…);` these two methods are responsible for retrieving one of the available free handlers from the read handlers pool and write handlers pool respectively, to handle the corresponding events on the channel. The scheduling, release, and memory parameters of these two methods are assigned to the retrieved handler before its execution is activated, by calling the `handle()` method, while the `selKeyToken` parameter holds the selection key associated with the event, this selection key holds information of the channel that received the event by holding an attachment object that is used as a token that holds the state information of this channel along the successive invocations of the handler.
- `monitorEvents();` this methods runs the monitoring loop within the component; e.g. polling for events, and it is called during the execution phase by the `runCommunicator()` method, this method internally calls the approporiate method for the defined mode, e.g. it calls `poll()` method for blocking mode, and `NBpoll()` for non-blocking.mode.

The third group of the operations is the group of the operations that are offered externaly to the user as defined in the `ICommunicator` interface, these operations include:

- `runCommunicator();` this method is called to start the execution phase of the component.
- `Initialize();` this method is called to start the initialization phase of the component.
- `setSelectorType();` is used to specify the type of the selector, as will be explained later.
- `setSelectorPolicy();` used to specify the selector policy, as will be explained later.

ClientCommunicator Class

This is a subclass of the communicator class that represents the client-side component. This class has no ability to accept requests from other clients; hence, it does not have Server Socket Channel, nor `acceptHandler` Pool; however, it can support one or more socket channels to make connections with remote servers, where these references of channels are held in the `chanelsQueue` that can be assigned a maximum number of channels that it can hold, through the `nMaxChannels` property, also, the actual number of channels in use is specified by the `nChannels` variable. Moreover, this class has a definition of a `connectHandlerPool` that holds the pool of handlers for handling the connect completion event, where the size of this pool is specified by the `nConnectHanlers` property.

This class also defines an internal method `fireNextConnectHandler()`, which can be used to fire a connection event handler, once the complete connection event occurs, using one of the free handlers in the `connectHandlersPool`.

Also, this class implements the following three methods of the `IClientCommunicator` interface to be offered to the user of the component:

- `createClientChannel(InetAddress remoteAddress, boolean isBlocking);` this method can be used to create a socket channel through which a new connection can be made to a remote server at the address `remoteAddress`, where this channel can be created to work either on blocking sockets or non blocking sockets according to the value specified by the `isBlocking` parameter.
- `makeConnection(InetAddress remoteAddress, boolean isBlocking);` this method internally creates a socket channel, using the `createClientChannel()` method, then it establishes the connection with the remote server whose, address is specified by `remoteAddress` property.
- `setCommunicationHandlers();` this method is used by the user of the component to specify the logic of the three communication events handlers, readHandlers, writeHandlers, connectHandlers.

ServerCommunicator Class

- This is a subclass of the Communicator class, and it represents the server-side component; hence, it should have the ability to accept requests from the clients. Therefore, unlike the `ClientCommunicator` this class has a `ServerSocketChannel` and `acceptorHandlerPool` that can have a maximum size of `nMaxAcceptHandlers`, whereas the actual number of the `acceptHandlers` is saved in the `nAcceptHandlers` variable. On the other hand, this class has no `connectHandlerPool`, as its socket channels are not used for initiating connections with other clients. This class has the following basic operations:
- `fireNextFreeAcceptHandler();` to start handling the acceptance of a new connection event, using one of the free handlers in the `accepttHandlersPool`.
- `getServerchannel();` this method returns the `SocketServerChannel` associated with this component.

B- The Communicator as a Separate Component

In order to build the Communicator component as a separate component, no big changes is needed to be made to the logic of the operations; the main changes are:

- The `Communicator` class has to extend the `ComponentCls` defined in the previous chapter in order to support all its facilities.
- The `Communicator` has to implement the `IBindController` if the component has to access other components.
- The class of the `Communicator` has to have a reference of the type `IMemoryModel` that can hold a reference to an object instance of the Forked Memory Model, and this in turn requires the creational operations to be changed to accept an instance of the `IMemortModel, e.g. ForkedMemoryModel` instance, instead of the memory context and its parameters. For example, to be able to create an instance of the client communicator, we can use the following creational operation:
- `createClientCommunicator(inetaddress:InetAddress, memoryModel: IMemoryModel, noHeap:boolean, synchPolicy:synchronyPolicyEnum, readLogic: Runnable, writeLogic:Runnable, connectLogic:Runnable): ClientCommunicator`
- If required, the Runnable objects: `connectLogic`, `readLogic`, etc. *may* be provided as instances of the `IStackLogic` defined in the framework, this requires no change to the interface definition because the `IStackLogic` definition already extends the `Runnable` interface.
- Instead of creating executors/handlers pools, the component can use the schedulable object pools(s) provided by the container, where it has to be configured during the initialization phase to have instances of the communication events' executors.

C- The 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 properties are important to be reconfigurable by the developer using the component:

- ***SelectorType;*** 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.

- ***SelectorPolicy***; by default, the selector in our design is working in a blocking mode, waiting for notifications of readiness of the communication events. However, it is possible to configure the Selector to work in non-blocking mode, i.e., the selector does not wait for the events, but it checks if there are any events have happened, and it returns immediately, even if there were no communication events to handle. Another possible configuration is to assign a Timeout T for it, so it can block waiting for communication events for maximum time length T_{block},, i.e. it returns if any event occurs before the elapse of the time T_{block},; otherwise, it returns once the time T_{block} elapses.

D- Executors (Communication Handlers)

The executors are the execution units within the component that are responsible for executing the code that handles the events occurring on the channels. In our model, we assume that these executors can be either normal Java threads, in case of using the component in non-realtime applications, or they can be schedulable objects as defined in the RTSJ, in case of using the component within a real-time application. Also, according to the component framework proposed in the previous chapter, these executors running in a real-time communicator component, whether the component is designed as a sub-component or a separate component, have to be reusable executors that can be provided by the schedulable/thread pools available in the container of the communicator component, in order to enhance the predictability of the execution and limit the concurrency.

Hence, according to these assumptions and based on the reusable schedulable object model presented in the previous chapter, we developed a class hierarchy for the executors; the class diagram of this hierarchy is shown in Figure 6-15.

As shown in the class diagram, we assume that the `Handler` class is an abstract class that implements the `IHandler` interface; the classes of the communication handlers are sub-classed from this class; whether it is real-time or non-real-time. The `Handler` class has the following set of basic attributes and accessible operations required for any handler:

- `initialize();` this operation is used to specify a server component object and a certain channel within this component, which this handler will be responsible for handling the communication events occurring on them.

- `getHandler();` this is an abstract method that has to be implemented by all subclasses representing a certain type of handlers, where each one of these subclasses is required to return the object reference of the executing element associated with it.
- `setHandlerLogic();` this operation can be used to assign a certain user-defined Runnable object to be the `logic` member of this class, where the `logic` member defines the logic to be executed by this handler.
- `handle();` in our model, we assume that the handler is created and ready for execution; however, it does not start to execute the logic associated with it, until an event occurs and this handler is chosen to handle it. So, the class provides the `handle()` method to be responsible for starting the execution. In this class this method is an abstract method, as each type of the handlers can have its own method to start handling the events.
- `Server;` this is a reference to the server component object itself that contains this handler, this reference enables the handler to access other elements of the component.
- `Channel;` this is a reference to the selectable channel that the handler is responsible for handling the events occurring on it.
- `theBuffer;` this is a reference to the user space buffer that holds the raw bytes associated with the I/O operation, i.e. it holds the incoming and outgoing bytes going through the channel which this handler is handling its events.

According to the above, our component model can be used in different applications whether they are real-time or non-real-time; hence, a set of different classes are assumed to be sub-classes of the abstract class `Handler`. These two subclasses are:

The `Thread_Handler` *Class*

The class `Thread_Handler` represents the non-real-time handler, i.e. it encapsulates a normal Java Thread that is used to handle the events occurring on a channel of this component; if the real-time guarantees are not required for this handler. Hence, in this type the communication, event *handler* can be an instance of the class `java.lang.Thread`. This class has to override the `run()` method in a way similar to the pattern provided in the last chapter for the reusable schedulable objects, but with omitting all the real-time functions. This class provides the method `getHandler()` to get a reference to its internal thread. Also, the inherited abstract method `handle()` is implemented within this subclass, in order to call the `start()` method of the thread to start the execution of the event-handling code assigned to it.

The `rtHandler` *Class*

As we assumed in the last chapter, in order to provide more predictable execution of the handlers, the RTSJ schedulable objects should be used as the executing elements for handling the I/O on the channels; hence, to represent the handlers that use schedulable objects, the `rtHandler` class is provided as an abstract class that extends the *Handler* class and implements the interface `IReusableSchedulableObject`, which was presented in the previous chapter, in order to provide a wrapper class for the real-time handlers. In addition to implementing the methods of the `IReusableSchedulableObject` interface, this class has the following methods:

`createHandler(Class handlerType);` this is a factory method that is used to create the schedulable object of the real-time communication handler within the memory area, in which the currently executing thread is running.

`createHandler(Class handlerType, MemoryArea memArea); t`his is another factory method that is used to create and initiate the schedulable object in the memory area, where the parameters of this memory area are specified in `memArea`.

- `getMemoryContext();` gets the currently assigned memory area of the handler **.**

The abstract wrapper `rtHandler Class` has a set of subclasses, which are inherited from it to support different type of handlers, these classes are:

– **The** `rt_RT_Handler` **Class;** this class is used when an instance of the `javax.realtime.RealtimeThread` is used as the schedulable object of the `rtHandler` class. This class implements the abstract `getHandler()` method that is defined in the abstract `Handler` class, and it returns a reference to the real-time thread instance used by this handler. This class provides as well the `handle()` method, to enable the start the execution of the logic assigned to it.

The `rt_NHRT_Handler` **Class;** this class is a specialization of the `rt_RT_Thread` that it extends, as it uses an instance of the `javax.realtime.NoHeapRealtimeThread` as the executing element of the handler, instead of using a `RealtimeThread`, this is to guarantee the highest predictable real-time execution of the handler, as the handler will be able to interrupt the garbage collector at any time, without any wait for the garbage collector to finish its execution cycle. As a requirement of the `NoHeapRealTimeThread`, the memory context of this type has to be from a non-heap memory, i.e. the scoped memory or the Immortal memory, to not interface with the garbage collection.

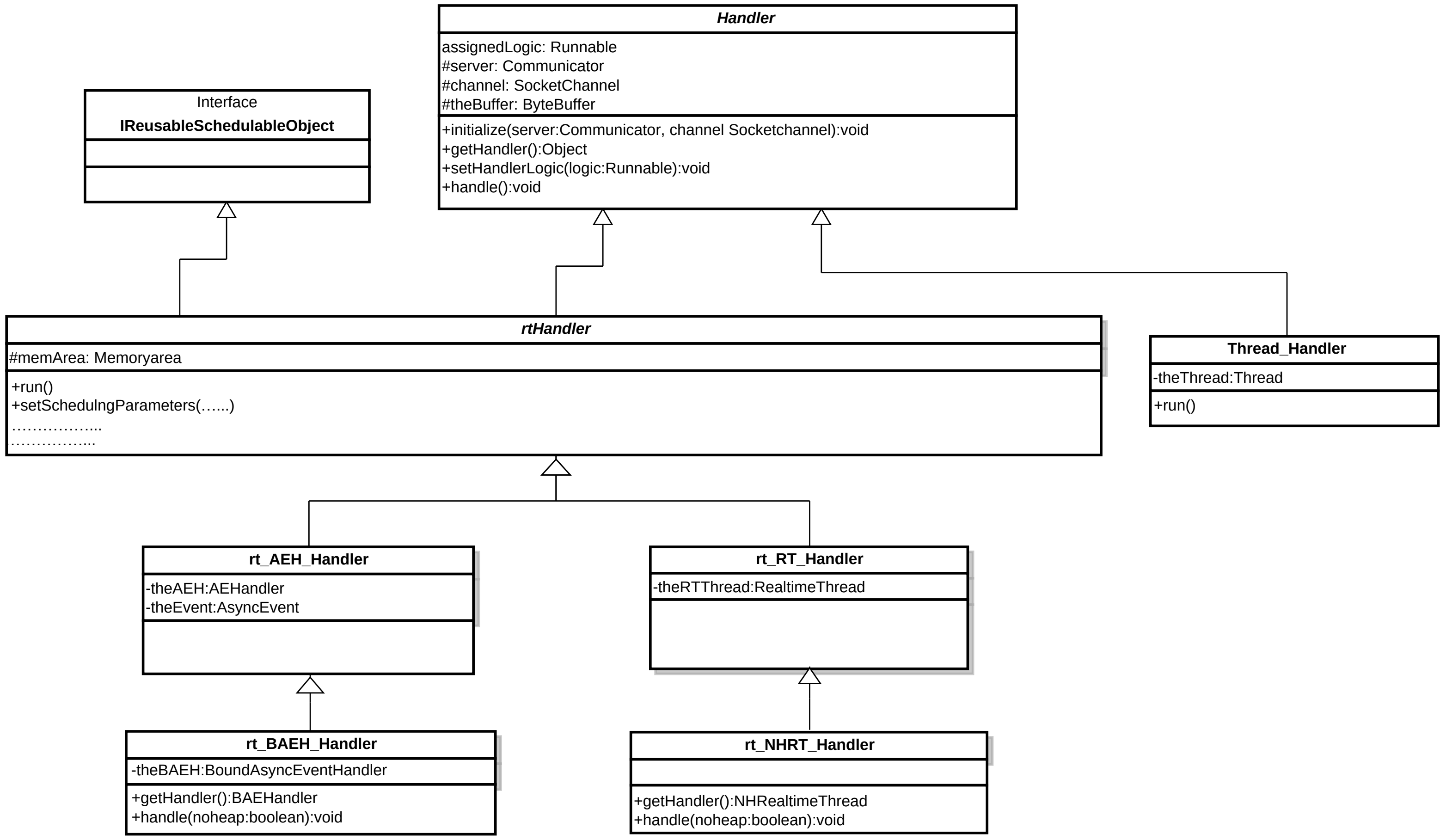


Figure 6-15 Handlers Class Hierarchy

– **The** `rt_AEH_Handler` **Class;** in this class, an instance of `javax.realtime.AsyncEventHandler` is used as the schedulable object executing the logic of the communication handler. The Asynchronous Event Handler runs when a certain event to which this handler is attached is fired. Hence, this class has a reference `theEvent` to an event object of the type `AsyncEvent`. The AEH starts execution by the dispatcher when the selector notifies the readiness of any of the I/O operations on the selectable channel associated with this handler. The dispatcher can fire the event and hence the handler by calling the `handle()` method defined in this class. Like other classes, this class provides the `getHandler()` method that returns a reference to the Asynchronous Event Handler used by this handler.

– **The** `rt_BAEH_Handler` **Class;** this class has the same structure as that of its super class `rt_AEH_Handler`, except that it uses an instance of the class `BoundAsyncEventHandler` instead of the class `AsyncEventHandler`. This is to ensure that the use of a dedicated thread to handle the event. This class can have a different form of the method `handle()` that accepts a `boolean` variable, to specify wheather the internal thread of this Bound Asynchronous Event Handler is a `NonHeapRealtimeThread`, or not. **Note:** In order to implement some of the patterns and examples presented in this thesis, e.g. in section 6.5 and appendix A.3, we used the class `EncapsulatedHandler`, as a class that extends the functionality of this class by supporting the reusability requirements mentioned in the previous chapter.

E- Event Handler Logic Runnable

This should be a class that holds the logic of services provided by the component; this logic is executed by executors, which in our model can be any schedulable objects. However, as proposed in (Pizlo, Fox et al. 2004), an executable logic running by schedulable objects can be represented in RTSJ as an encapsulated method. Hence, to be externally assignable by the developer, the developer writes his own encapsulated method and assigns it to this property of the component. In case of using the design for a separate component, the user can assign the logic either as any class that implements the `Runnable` interface, or as a class that implements the `IStackLogic` which itself extends the `Runnable` interface.

6.4.3 Component Lifetime

The main purpose of building the Communicator component is to provide a reusable component that integrates the RTSJ with the networking Java packages, in order to support several communication strategies that are commonly required by many real-time middleware. We have provided in the previous sections the internal structure of the component and the design of its internal classes. In this section, we define two different phases of the lifetime of the component and specify the operations that can be executed during these phases; these phases are *the Initialization phases*, and *the Execution phase*. The scenarios of operations occurring during each one of these two phases are presented next, where we discuss how the framework presented in the previous chapter can support the implementation of these phases.

A- Component Initialization Phase

In this phase, objects that have to be available for the lifetime of the component are created and initialized according to the assigned configurable properties, in order to make the component ready for the processing in the next phase; the execution phase.

Separating the initialization phase from the execution phase is a common technique in many real-time systems, as the process of object creation and initialization is a heavy process (due to the need of loading Java classes); hence, this process is required to be done only once. On contrary, the operations executed during the execution are always timely constrained, and they may be executed repeatedly. So, the execution phase operations should not interface with any setting-up operation, in order to have a more predictable execution phase of the component. In the following, we present a list of the objects created and initialized during the initialization phase:

The proactor dispatcher*;* this is the basic element of the component, it is created in this phase, where, in case of the server-side component, it uses the component's configured I/P address to create internally an instance of the `ServerSocketChannel` class that monitors the incoming requests to the server. Then, the proactor dispatcher is set to run either in one of two modes; the blocking mode (in case of multi-threaded server), or the non-blocking mode (in case of proactive, or reactive server). Also, in this phase, the *proactor dispatcher* registers with the selector the appropriate events, which are required to be handled by the component, i.e. it registers the required read, write, and connect events, in case of the client mode, while it registers the required read, write, and accept events, in case of the server mode.

Also, in this stage, the *Selector* is created, where its creation is made by using one of the sub-classes of the Java NIO `Selector` class, which represent the configured selector type that this component is configured to use it. 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, by calling the method `Selector.open()`.

Also, if the component is configured to have thread pools, then the executors' pool are first created; then, these pools are filled with the specified number of objects of the configured Communication Handlers' type. Where all these communication handlers are created in this phase as passive objects, i.e. they are not actively running or scheduled for execution.

During the initialization phase, the byte buffers, which are used to hold the bytes sent/received during the I/O operations, are created. These byte buffers can be created either as *direct* or as *indirect,* as mentioned before in section 6.2.3-A-. This raises a question that requires an answer; *are the direct and indirect buffers compatible with the RTSJ memory model? and do we need both types with RTSJ?*

To answer this question, we first consider the direct buffers; like any normal Java objects, the direct buffers are created in the heap; i.e. they are collectable by the garbage collector. So, as any Java objects, the allocation of any of these object in any of the non-heap memory areas defined by the RTSJ results in a better predictability of the non-direct byte buffer object, as it will not be interfered with, or moved by, the garbage collector. However, there will be no chance of avoiding the copying process from the operating system buffers to the inner buffer of the byte buffer, as the inner buffer of the non-direct byte buffer is allocated within the address space of the memory area that contains it.

Now to answer the other part of the question, we consider the direct byte buffers. In normal Java, the direct buffer is created in the heap as a normal Java object, with a reference to a buffer that holds the bytes of the I/O operation out of the heap. The same object can be created in one of the memory areas defined by the RTSJ, but the question now is about the inner buffer that is created out of the object's memory area and will be referenced from it. The RTSJ memory assignment rules are a set of rules that defines the possibilities of referencing an object within one memory area from another memory area. In the case of direct byte buffers, the external memory allocated for the byte buffer contains only raw bytes; hence, there are no way to access any object within it, and the actual access is to the byte buffer object itself, which in turn accesses the bytes of the external memory. However, the RTSJ provides the class `javax.realtime.RawMemoryAccess`, to model a range of memory as a sequence of bytes; hence, in our model we assume that the creation of the native inner buffer can be made in a physical memory, which can be accessed using an instance of the `javax.realtime.RawMemoryAccess.`

So far, we discussed the basic objects to be allocated in this phase. But, to create these objects, in case of using real-time Java, we need to specify where to allocate these objects. To specify where these objects should have to be created, we have to consider our component model that we provided in the previous chapter. In this model, we assumed the existence of a memory area for each component that keeps objects created within it alive along the lifetime of the component; we called this memory area as the *Component Memory Area,* and considered it to be the base memory area within the component containing it. Hence, we consider that the creation of all objects that are created during the initialization phase of the communicator are to be in the component memory area of the communicator component. So, the component's memory area should be assigned as the *memArea* parameter of all the methods and constructors used to create these objects.

B- Component Execution Phase

The execution phase of the component is assumed to provide the basic functionality that this component provides. So, in our communicator component, once the communicator starts its execution phase, it enters an infinite loop to handle the events occurring on the component's channels. In this section, we discuss the execution phase of the Communicator Component, when it is used as a server side component; this does not mean that the client-side Communication component has a complete different execution phase, as the execution phase for both is identical except for the kind of events and operations that is required for each of them.

In the execution phase, when any of the registered events occurs, the *proactor dispatcher* retrieves a reusable free executor from 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 for executing the *Handler Logic Runnable* corresponding to the completion handlers of events occurring at the channels. Before starting the execution, the executor parameters have to be retrieved and initialized; these parameters are retrieved either from the parameters tables, if the component is configured to use server centric approach, or it is retrieved from the client and assigned to the executor dynamically.

The processing within the execution phase of the Communicator Component that works at the server side starts once a client requests a connection to this server, at this moment, 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 the client. After accepting the connection, the server component behaves differently for each type of the supported server types, as each type involves different operations for receiving, and handling the client requests. In the following, we discuss the different operations for each of the three defined server-communicator models.

I- NonBlocking Proactive Model Execution Phase

In this non-blocking proactive model, see the sequence diagram in Figure 6-16, when a connection request arrives to the server; the operating system notifies the selector of this event; the selector in turn notifies the proactor dispatcher, and forwards to it the event information, including the created socket channel object (client channel) responsible for 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 the *read* event with the selector, in order to be notified when any read operation is made through this channel. Then, the proactor dispatcher retrieves a free executor from the executors' pool, i.e. an *acceptor handler*, in order to be a handler for the accept event. As the acceptor handler executor runs 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, the scheduling and the release parameters of the acceptor executor can have either default values, or they can be loaded from static tables, i.e. only the server centric approach is supported for the acceptor handlers. One possible task that can be processed by the acceptor handler is to retrieve the execution paramters from the client, in order to assign them to the handlers of the next client requests, i.e. support the *client propagated parameters* approach in this model.

After that, 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, as long as the read operation is registered with the selector; to handle this event, the proactor dispatcher retrieves another free executor from the executors' pool; however, this time the executor can be assigned either server centric parameters, as the acceptor executor, or use the client propagated parameters, if they have been retrieved by the acceptor handler. Then, the executor starts executing the configured *Reader Handler* of the component.

In this non-blocking model, every time a read event arrives, an executor is retrieved from the executors' pool; hence, there is no guarantee that the same executor is used to handle all the read events on a certain channel. This requires a mechanism to ensure that the activated handler is assigned the required logic, where this logic has to be the same one used by all the read executors of a certain channel; in other words, all the read executors of one channel during one session needs to be assigned the same Runnable class, where each executor runs only its assigned part of the logic defined in this class. One possible way to provide this functionality is to define the logic class as a state machine, and assign for each read handler a certain state within this state machine to execute, this is dicussed in more details in the example presented in section 6.5.

On the other hand, as the write event is fired any time the channel is ready to write to; hence, if the write handler has to be executed, the write operation is registered with the selector only just before starting to write to the channel. So that, 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 assigns its parameters in the same manner as in the read event. However, in this case the executor executes the logic defined in the Writer handler runnable.

A prototype implementation of the `pollNB()` method, which provides the above operations in the non-blocking mode for observing and handling of the network events in the communicator component, is presented in appendix A.9.

II- Non-Blocking Synch Reactive Model Execution Phase

The execution phase of this model, see Figure 6-17, behaves initially the same as the proactive model, i.e. the dispatcher is notified of the incoming events on the registered server channel, where this channel is running in a non blocking mode. The major difference between the reactive and proactive model comes in the way of executing the handlers. The reactive model is assumed to be single threaded; this means that when a registered event occurs on the client channel, instead of retrieving another executor from the executors' pool to execute the logic assigned to the required event handler, the proactor dispatcher itself acts as the executor of all the handlers' logics, i.e. the logic to handle the registsred events can be internal methods of the communicator class, or it can be defined as Runnable objects, where the proactor dispatcher has to enter the scoped memory assigned for this handler, and runs the required logic inside it. The scheduling and the release parameters used in this model can be retrieved in the same way, as described above in the proactor pattern. However, as only one executor is running in this model, the scheduling and release parameters of this executor may have to be changed frequently and dynamically each time a new event arrives into the system.

III- Synch Multithreaded Model Execution Phase

This model can be implemented either directly, using blocking socket channels, or indirectly, by using the non-blocking proactor dispatcher model to emulate it. To implement it directly, see the sequence diagram in

, the socket channel created to handle client connection request 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 have to be directly monitored and handled by the executors. Therefore, in this mode, the proactor dispatcher works as a connection listener; once a connection is requested, the dispatcher retrieves a reusable executor for this client connection from the executors' pool. Once this executor starts to run, all the events on the channel are directly executed by it; hence, the executor has to be responsible for invoking and running all the communication logic runnables, i.e. acceptor handler, reader handler, write handler.

On contrary, to emulate the work of this reactive multithreaded synchronous model, the server channel, see the sequence diagram in Figure 6-19, is configured as non-blocking, but the responding to the events is handled differently. The main difference is that instead of creating/retrieveing a new executor to handle each new event occurs on the client channel, only the acceptor handler, which is the executor created when the accept event occurs, blocks waiting for notifications to continue processing the next event(s). This can be implemented by attaching this acceptor executor, as a token when registering any event(s), during the execution of the logic attached to this acceptor executor. Then, when any of these registsred events occurrs on this client channel, a notification is made by the operating system to the selector to activate the event handling operation; to handle this event, the selector retrieves the data associated with the received event, and extracts the token attached with it, i.e. the acceptor executor; then, the this executor, which is blocked waiting, is notified to wake up to continue the execution of the logic attached to it. Hence, in this mode, only one executor with one Runnable is used during the session of communication between the client and the server.

A prototype implementation of the `poll()` method, which provides the above operations in the Emulated-blocking mode for observing and handling of the network events in the communicator component, is presented in appendixA.10.

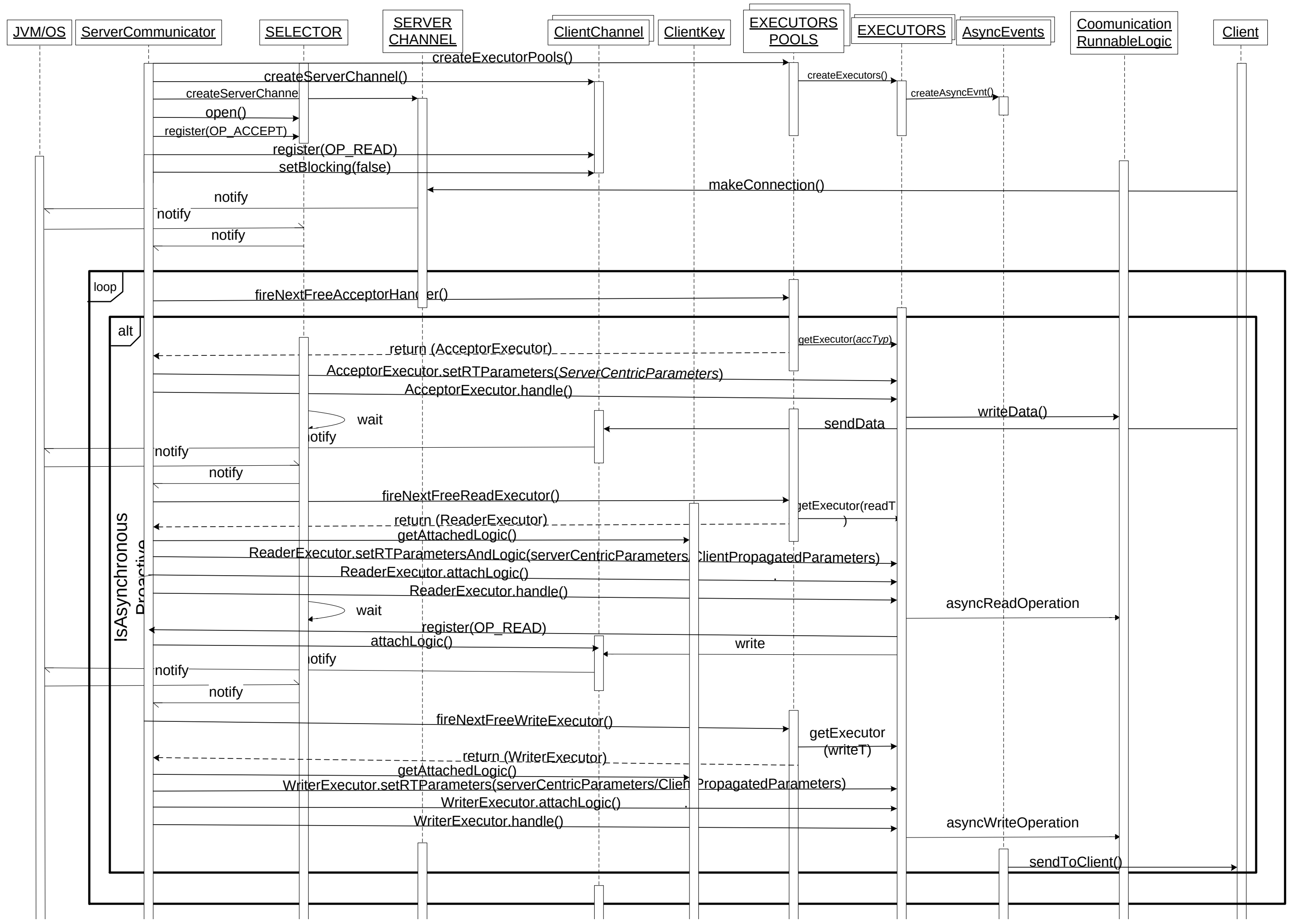


Figure 6-16 The Non-blocking Proactive server model

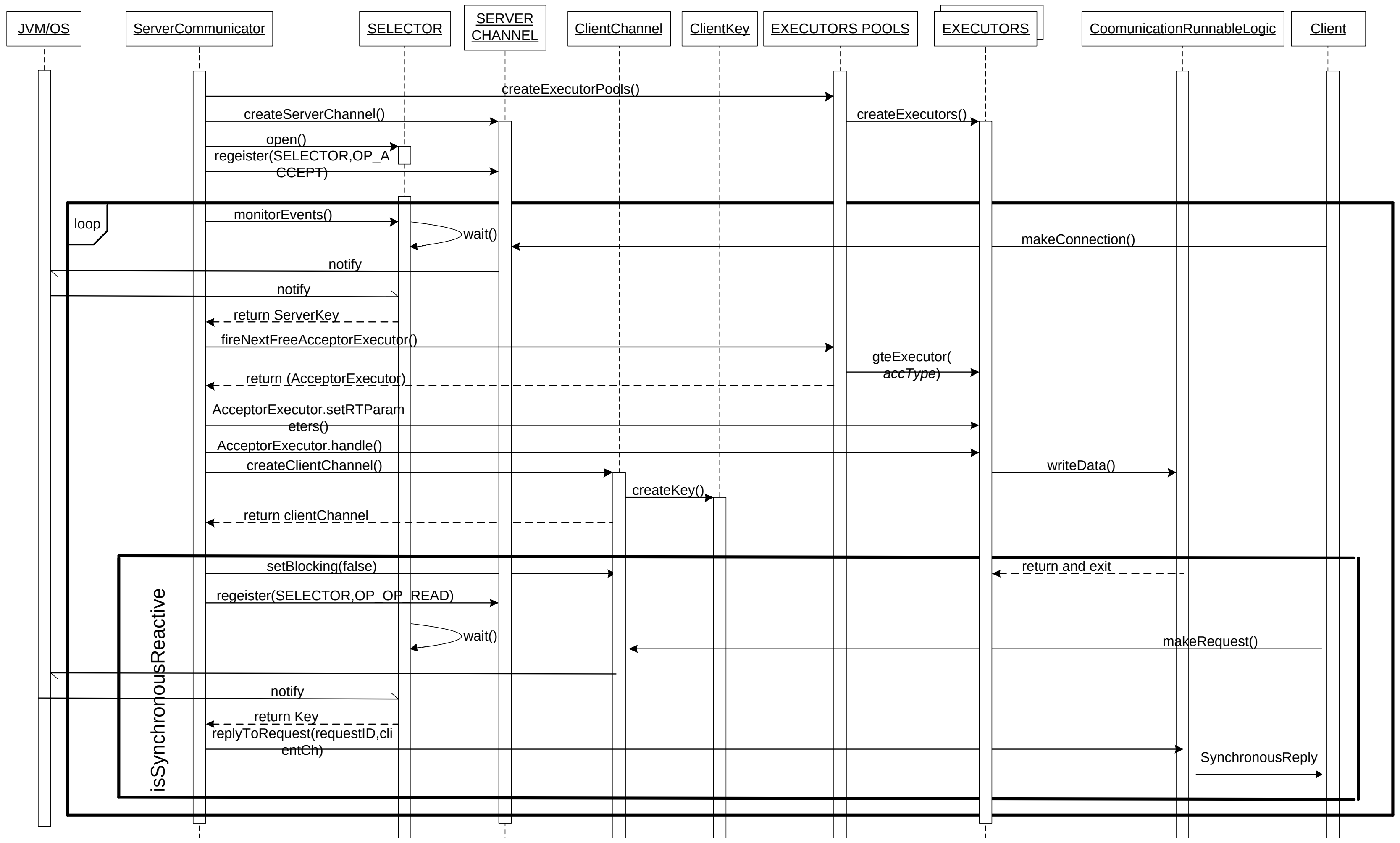


Figure 6-17 Synchronous Reactive Mode

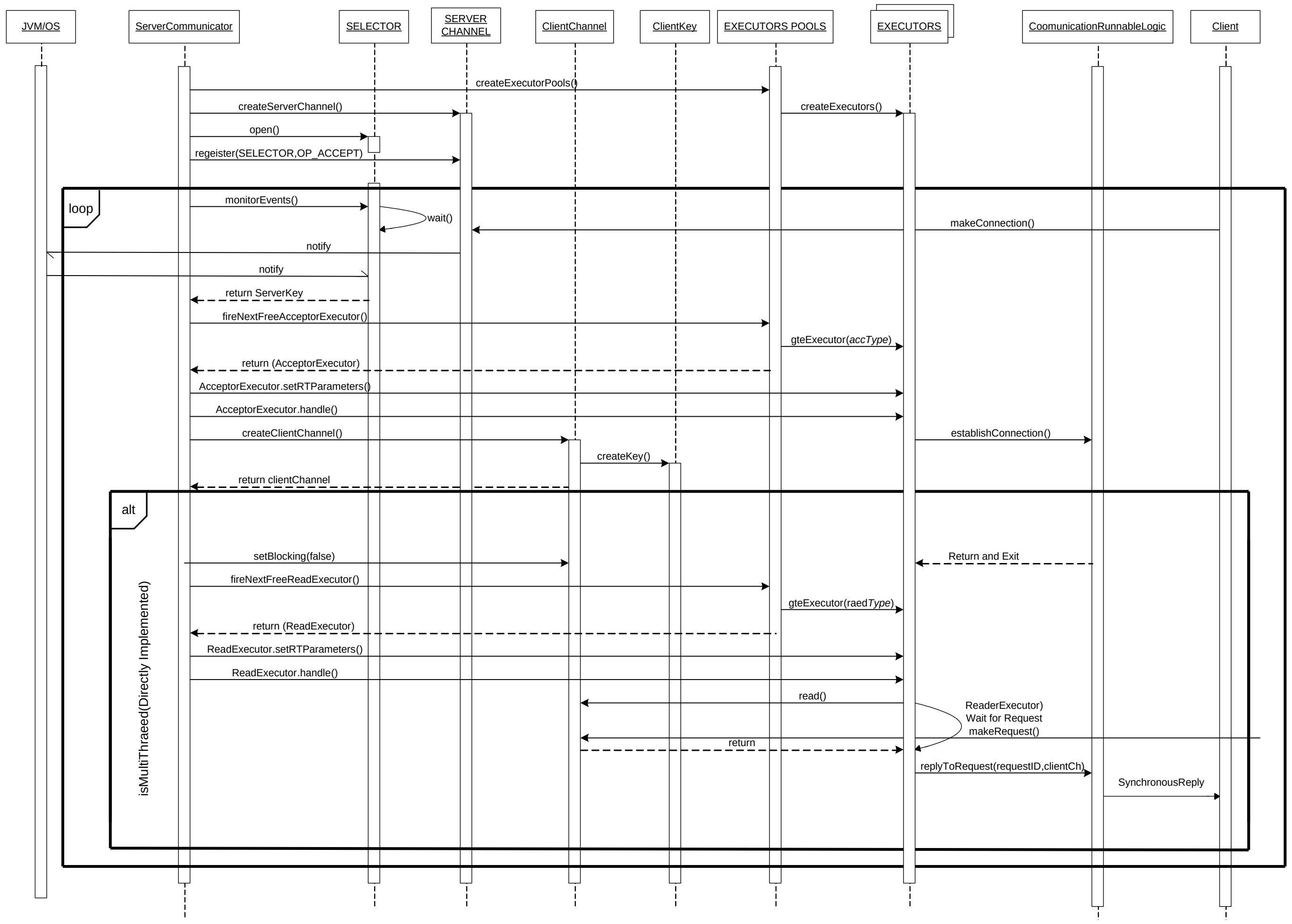


Figure 6-18 Multi-threaded-Direct Implementation Mode

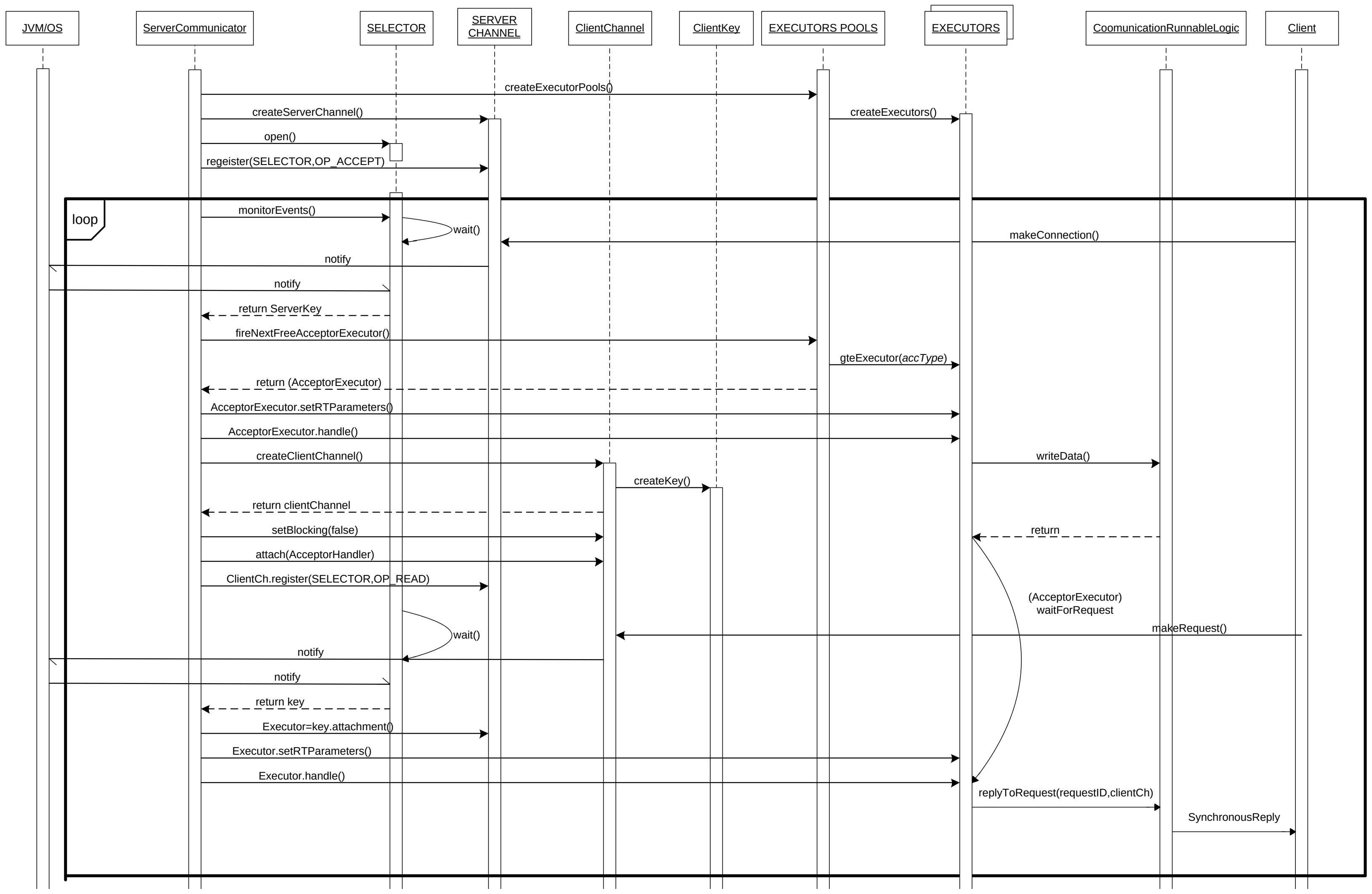


Figure 6-19 Synch-Multithreaded – Emulated Design Mode

6.5 Using the Framework in a Client-Server Application

In Chapter 5, we presented the set of RTSJ-based models and design patterns. These models and patterns, in addition to the communicator component presented in this chapter, form the basic elements of our proposed component framework. In this section, we present an example that shows how these elements can be integerated together to build a simple client-server application, which supports the low-level remote communication, i.e. sending and receiving bytes.

In our example, we use a very simple scenario of a client that sends periodically a set of bytes to the remote server, which replies to the client by sending these bytes back to the client. The diagram shown in Figure 6-20 shows our proposed structure for this simple application. The diagram has the following elements:

1- **ClientSide Container**. This container has two main elemnts; the caller component that makes the call, and the communicator component which exchange the bytes with the server.

2- **ServerSide Container.** This container has two main elemnts; the communicator component which exchange the bytes with the client, and a server component.

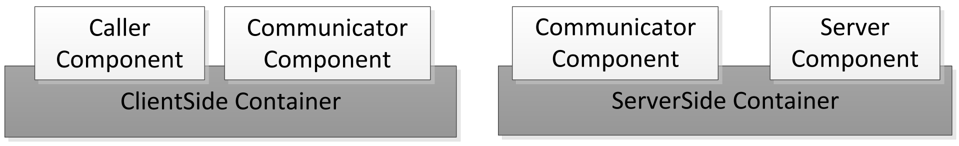


Figure 6-20 Example Structure

In the following, we present an overview of how our framework is used to build these elements.

6.5.1 The Client Side

In the implementation of our example, the container at the client side was implemented in the RTSJ class `ClientSide`, which extends the `ContainerCls` that represents the container model in our framework. In the `ClientSide`, the `BuildComponent()` method was oveerideen to define the inner components of this container. This overridden method, starts with a definition of a scoped memory area to be a common component memory area of all the components in this container using the statement:

```
LTMemory theCMA = new LTMemory(2000000, 3000000);
```

Then, the caller component is created in four main steps:

```
Caller client = (Caller)new Caller();//instance of the comp
client.init(theCMA, 100000, LTMemory.class, isizesServer,
       msizesServer, DualFork.class);//initialithe the comp
client.setComName("Client");//assign a name
addComponent(client);//add to the container
```

In these steps, an instance of the component is created from the component class Caller. Then, the component is initialized to use the common memory area, and aqwwigned the `DualFork` class to manage the life time of any internal SMAs within this component. Then the component was given a name, and finally the component is added to the container's components.

As we proposed in the scenario of our example that the component has to be periodic, then, in the next step, a periodic task is created to run within the component.

```
ReusableRunnableStack task = client.addPeriodicSMATask("ClientExecutor",
RealtimeThread.class, null,
      new PeriodicParameters(null, new RelativeTime(1000, 0)), 10000,
10000, LTMemory.class, null,
      ClientLogic.class);//Create a a periodic SMA task
task.setParameters(LTMemory.class, 3, new LTMemory[] {theContMA, theCMA},
initM, maxM, client);//assign the paremeters of the task
```

This task was added as periodic SMA task, which is a realtime periodic thread with a period of one second and has its scoped memory as LTMemory and runs the logic defined in the `ClientLogic` class. The creation of this task returns an instance of a class that represntes the Reusable Runnable Stack pattern, this instance represents the runnable stack created by the framework to this added SMA task. In our example, this reusable runnable stack is configured to have three levels of scoped memory areas; the container memory area, the common memory area, and the third level is a new temporary scoped memory area of the LTMemory area type; the initial and maximum memory sizes of this memory area is assigned, and the reusable runnable stack is assigned the client component as its parent component.

The creation of the other component, i.e. the communicator component, starts with similer steps, shown next, with the parameters of this component. For example, the logic of the communicator component is specified to be taken from the `CommunicatorAsClientLogic` class.

```
  CommunicatorCls communicator = new CommunicatorCls();//create the Communicator
Component
  communicator.init(CMA, 10000000, LTMemory.class, isizesComm, msizesComm,
DualFork.class); //initialize the communicator
  communicator.setComName("Communicator");//Give the name
  addComponent(communicator);//add  to this container

  ReusableRunnableStack clientLogic =
      communicator.addSMATask("ClientCommunicator", RealtimeThread.class, null,
null, 10000, 10000, LTMemory.class,
          null, CommunicatorAsClientLogic.class);//add a new SMA task to the
Communicator
  clientLogic.setParameters(LTMemory.class, 3, new LTMemory[] {theContMA,
theCMA}, initc, maxc, communicator);//set  task parameters
```

However, as the communicator component has its own properties that define its behaviour, so, another set of statements are added, shown next, to define these properties. The first statement defines the sizes of the handlers pools, then, the next thress statements specifies the logic of each of the Connect, Read, and Write event handlers to be the classes `ConnectLogic`, `ReadLogic`, and `WriteLogic` repsectively. Finally, the last statement specifies the mode of operation of this component to be the client-side mode of operation.

```
communicator.createHandlersPool(20, 120, 120, 20);
communicator.setConnectLogic(ConnectLogic.class);
communicator.setReadLogic(ReadLogic.class);
communicator.setWriteLogic(WriteLogic.class);
communicator.createObserver(1);
```

The `BuildComponents()` method ends with a call to the `start()` method to start the execution of the components within the client side container.

As we saw above, the creation of the client side involves the usage of a set of classes other than the container, these classes are:

- `CommunicatorCls` class, and `Caller` class: The classes of the components
- `CommunicatorAsClientLogic` class and `ClientLogic` class. The classes of the logic of the SMA tasks running within the client and the communicator components
- `ReadLogic`, `WriteLogic`, `ConnectLogic` classes. Define the logic of the events handlers within the communicator component.

In the following, we present an overview of the logic defined in those classes.

A- CommunicatorCls

This is the class that represents the Communicator component; as presented in this chapter, this class extends the framework's Component class with the functionality required to support low level remote communication.

B- CommunicatorAsClientLogic

This class in our example is responsible for defining the logic of an inner task within the communicator component; this inner task initiates the execution of the component, by calling the `runAsClient()` method of the communicator component. In our example, this class implements the `IStackLogic` interface, in order to make this class the stack logic component that runs within the created SMA task. The call of the `runAsClient()` method is made from within the `runUpward()` method once the task enters the memory level (0) of the scope, i.e from within the container memory area, where the access to the communicator is made through the `parentComponent` argumengt of this method.

C- Caller class

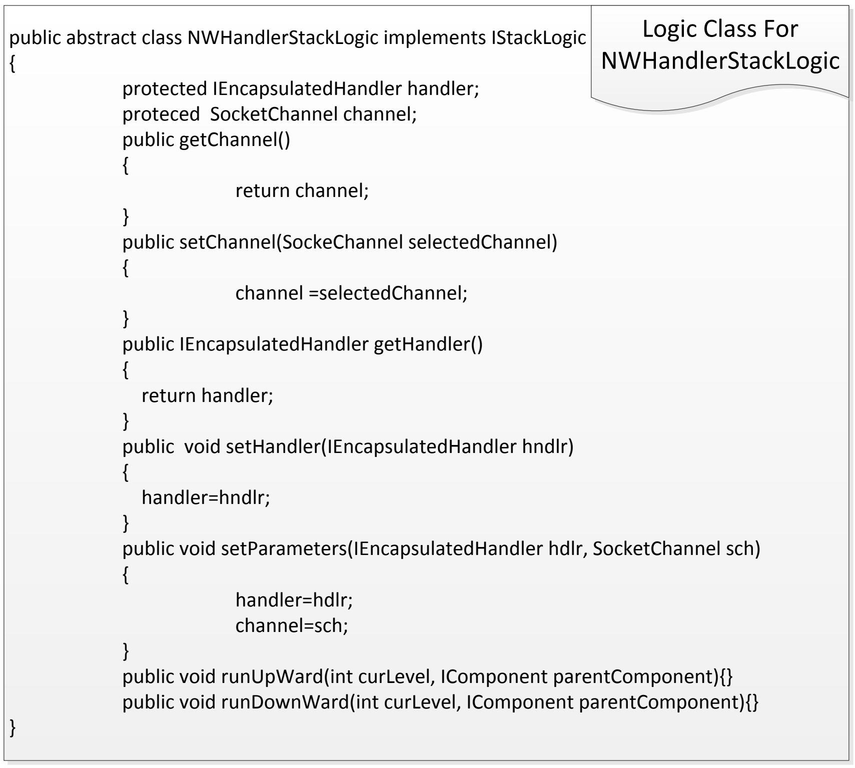
In our example, the caller component has a very simple architecture, as it acts as a holder to a single periodic calling task, so the `Caller` class just extends the `ComponentCls` that represent the component model, and it has no extra functionality.

D- ClientLogic

The `ClientLogic` class in this example holds the logic of a periodic task which runs within the client component, this logic is responsible for creating a packet and sending it over the network to a remote server; then, it waits to receive a reply packet from the server side, and this process repeats periodically. As this class is doing this process remotely over the network, then it uses the Communicator component to process the remote communication operations. So, there are integeration of the work of this class with the `CommunicatorCls`, which has the implementation of the Communicator component; the Communicator component, as presented in this chapter, supports more than one mode of operations, e.g. blocking, and non-blocking. Therefore, in order to see the differences in the programming models of these modes, we present next the implementation of the `ClientLogic` class in our example in two different modes: the emulated blocking, and the non-blocking mode.

1- The Emulated-Blocking Mode

In this configuration, the `ClientLogic` class extends the `NWHandlerStackLogic` class, shown in Figure 6-21; the `NWHandlerStackLogic` class implements the runUpWard() and runDownWard() methods of the IStackLogic interface in order to build a Reusable Runnable Stack pattern component, i.e. to support the execution of the logic on a stack of scoped memory areas as presented in chapter 5. In addition to that, the NWHandlerStackLogic class defines a set of methods and parametrs specific to the network event handling required for the communicator component. It defines the handler as a reference to an instance of the the event handler provided by the communicator component, also, this class defines a reference to a network channel, which is the channel whom the events occurred on it are handled by the defined handler. Associatred with these two references, the NWHandlerStackLogic provides set and get accessor methods to access these references. In the following, we present the implementation of the ClientLogic, and the AcceptLogic classes.


Logic Class For
NWHandlerStackLogic

```
public abstract class NWHandlerStackLogic implements IStackLogic
{
            protected IEncapsulatedHandler handler;
            proteced  SocketChannel channel;
            public getChannel()
            {
                        return channel;
            }
            public setChannel(SockeChannel selectedChannel)
            {
                        channel =selectedChannel;
            }
            public IEncapsulatedHandler getHandler()
            {
              return handler;
            }
            public  void setHandler(IEncapsulatedHandler hndlr)
            {
              handler=hndlr;
            }
            public void setParameters(IEncapsulatedHandler hdlr, SocketChannel sch)
            {
                        handler=hdlr;
                        channel=sch;
            }
            public void runUpWard(int curLevel, IComponent parentComponent){}
            public void runDownWard(int curLevel, IComponent parentComponent){}
}
```

Figure 6-21 The NWHandlerStackLogic class

The ClientLogic class itself has references for two buffers, inbuffer, to hold the incoming packet, and outbuf to hold the output buffer, in addition to that it defines references to the Communicator, the the localhost, the remote address of the server, and its port number.

As the ClientLogic class represents a reusable runnable stack logic component, in which the handler executes the logic defined in the runUpWard() method for each memory area it enters in the stack, i.e. in the *UpWard* propagation, and then it executes the runDownWard() method before exiting each of these same memory areas, i.e. in the

DownWard propagation. In this section, we present the logic of these two methods when the communicator component is configured to run in both the Emulated-Blocking mode, and the nonBlocking mode.

The logic provided by this class, shown in Figure 6-22, is executed on the three scoped memory levels defined for the client component, i.e. the ContainerMA of the container holding it in level [0], the CMA of the component itself in level [1], and the temporary scoped memory area in level [2]. In the following, we present the how we implemented our example within this these different memory areas in a way that enables the integeration with a communicator component which is configured to run in the Emulated-blocking mode.

- **IF [UpWard], i.e. propagating upwards**

⇒ If [curLevel==0], i.e. entering the ContainerMA:

In this level, a check on the communicator component reference is made to see if it has been assigned or no, if it has not been assigned, then the reference is retrieved from the set of the internal components references of the current container. Then, a connection is made with the target remote server. Using the `makeConnection()` method of the communicator component.

The check is important because this code is rexecuted for each period; so, the operations of retrieving the communicator reference and making the comnnection will be made only in the first period, and would not be repeated for each period.

⇒ If [curLevel==1], i.e. entering the CMA:

The CMA is the main memory level of the component, i.e. it is the memory in which the main functionality of the component is processed. In our example, the the main functionality of the client component consists of two parts; sending the packet to the server, and then receiving the reply. The first part is done in this step, i.e., once the CMA is entered, whereas the other step is done in the downward propagation, i.e. before exiting the CMA memory area. The code of the first part involves the clear of the output buffer, filling it, where in this example we fill it with random values; then, flipping the buffer to move the cursor to the beginning, and finally writing the contents of this buffer to the channel.

⇒ If [curLevel==2], i.e. entering the temporary scoped MA:

The code executed in this level contains a set of printing statements of logging some data of the processing, such as the number of sent packets so far. As the `String` class in Java is immutable class, then joining two or more strings in the Java language results in the creation of hidden objects. So, it is important to run this code here to ensure that these created hidden objects are reclaimed once this memory area is exited; otherwise, if this code was written in the level (0), or level (1), this would result in accumulating these objects in these memory areas, which may result in memory leak of them, as the code in this example is executed periodically.

- **IF [DownWard], i.e. propagating downwards**

⇒ If [curLevel==2], i.e. leaving the temporary scoped MA:

No code is provided here, as this part is just executed before leaving the temporary memory area, i.e. in the same memory area of the last step, so the same code can exists in one of them or divided between both of them.

⇒ If [curLevel==1], i.e. leaving the CMA:

Here, in the CMA, we execute the second part of the component functionality described earlier, i.e. receiving the reply packet from the server. This involoves the following operations:

- Add the communication channel used by this component to the registeration queue, where it is saved with the required operation OP_READ, and the input buffer, as an attachement.
- Send activation signal to the selector to activate it; this enable the selectorto do the registeration of all the saved channels in the registeration queue, including the one just added in the last step; so that, the selector can be able to observe the registered events of these channels.
- The current handler is enforced to block waiting for notofications from the component, in order to start the processing the receiving of the incoming packet.
- Once, the handler is notified by the selector of the arrival of bytes on the channel, the handler is activated and starts to read the data; this step with the last step are enclosed in a single loop that continues as long as the number of received bytes are less than the expected number of bytes defined for the input packet.
- After the packet is completely received, the packet contents can be parsed and decoded, e.g. by calling a method like `decodePacket()`.

⇒ If [curLevel==0], i.e. leaving the ContainerMA:

The handler arrives to this step after exiting the component memory area; hence, before finishing its execution cycle, the logic has to be prepared for the next execution cycle in the next period. So, all reusable values/object can be reset, and the files/channels that need to be closed can be closed in this step, e.g. in our example, we reset the value of the numer of bytes in this step.

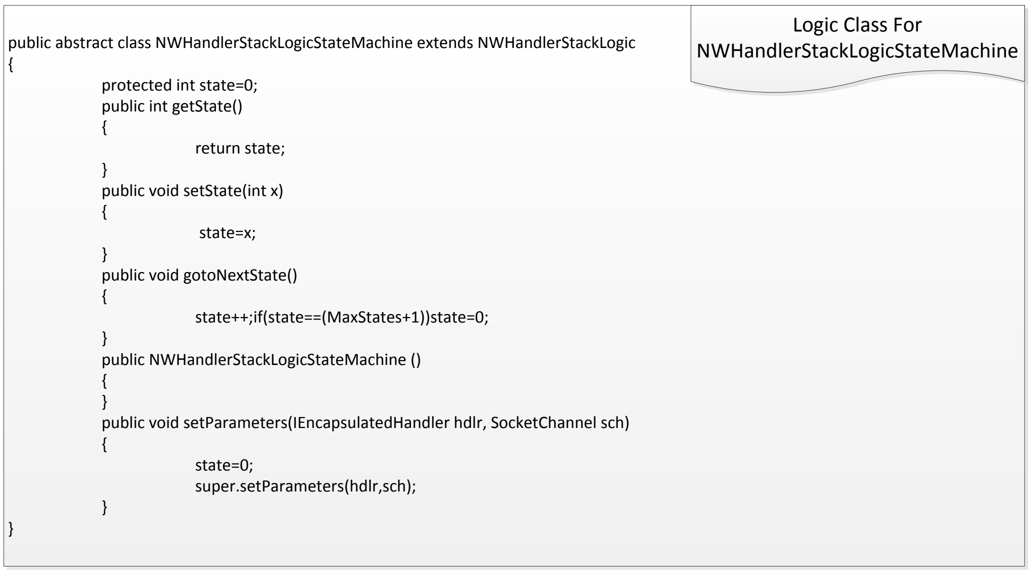
```
public void runUpWard(int curLevel, final IComponent parentComponent) {
  if (curLevel == 0) { //---=>>>runs in the container memory area
   if (communicator == 0) {
    rc = Clock.getRealtimeClock(); //get the rt clock
    Random aRandom = new Random(); //create a random number generator
    vvv = (int)(aRandom.nextInt(10) + ((rc.getTime().getNanoseconds() / 1000) % 10))
* 1000; //Get a random value
   }
   communicator =
((CommunicatorCls)(parentComponent.getContainer().getComponents().get(
      "Communicator"))); //Get a refrence to the communicator
   channel = communicator.makeConnection(remoteAdress, 2190, 25); //connect  }
  if (curLevel == 1) { //runs in the component memory area
   bufout.clear();          //clear the buffer
   for (int i = 0; i < 10; i++) {
    bufout.putInt(i + vvv); //fill the buffer with random numbers
   }
   bufout.flip();//flip the buffer to be ready for the write operation
   channel.write(bufout); //write the bytes from the buffer
  }   //the component memory area
  if (curLevel == 2) {//the temporary scoped memory area
   //The following code displays messages to the user
   System.out.println("The Packet Number " + packetNumber++ + "Has been written");
   System.out.println("The client started to wait At" + tstart);
  } //end of curlevel==2
 }  //end of runUpward

 public void runDownWard(int curLevel, IComponent parentComponent) {
  if (curLevel == 2) { }
  if (curLevel == 1) { //runs in the component memory area
   bufin.clear(); //clear the output buffer
   try {
    do {
     try {
      communicator.registerationQueue.add(channel, SelectionKey.OP_READ,
          bufin); //add the channel to the registeration queue
      communicator.theControllerChannel.sigQueueToSignalFD(12,
          10); //send the interrupt control signal to enable the registeration
     } catch (Exception e) {
      System.out.println("Exception....." + e);
     }
     bufin.clear(); //clear the output buffer
     try {
      synchronized (getHandler()) {
       getHandler().wait();//wait for the notification of packet arrival
      }
      nn += channel.read(bufin); //read from the channel into the output buffer
     } catch (Exception ex) {
      System.out.println("*----exeption-----*" + ex);
     }  //end catch
    } while (nn < 1000); //read upto 1000 bytes from the output buffer
   } catch (Exception m) {
    System.out.println("---exeption---" + m);
   }
      bufin.flip(); //flip the output buffer
   for (int i = 0; i < 250; i++) {
    try {
     int m = bufin.getInt(); //read the next integer value from the buffer
    } catch (Exception nb) {
     System.out.println("-Exception-" + nb);
    }
   }
  //Code for Processing the reply packet
  }
  if (curLevel == 0) { //runs in the container memory area
   //RESETTING THE VALUES and CLOSING any opened files, etc. if noyt needed any more
   nn = 0; //reset to initial values to be recyled clean
  }
 }
```

Figure 6-22 ClientLogic [Emulated-Blocking]

2- Non-Blocking Mode

In the configuration of the non-blocking mode, the `ClientLogic` and the `AcceptLogic` classes are required to work on the principle of using a state machine to enable the execution of the same logic by several handlers, where each handler executes the logic in one of its states. Hence, to satisfy this requirement, these two classes extend the `NWHAndlerStackLogicStateMachine` class that extends the `NWHandlerStackLogic` class to support a very simple state machine. This class, shown in Figure 6-23, defines a state variable that defines the current state of the logic, and defines methods to manage this state, e.g. `gotoNextState()` is used to forward to the next state.


Logic Class For
NWHandlerStackLogicStateMachine

```
public abstract class NWHandlerStackLogicStateMachine extends NWHandlerStackLogic
{
        protected int state=0;
        public int getState()
        {
                return state;
        }
        public void setState(int x)
        {
                 state=x;
        }
        public void gotoNextState()
        {
                state++;if(state==(MaxStates+1))state=0;
        }
        public NWHandlerStackLogicStateMachine ()
        {
        }
        public void setParameters(IEncapsulatedHandler hdlr, SocketChannel sch)
        {
                state=0;
                super.setParameters(hdlr,sch);
        }
}
```

Figure 6-23 The NWHandlerStackLogicStateMachine

The logic defined in this class, in the non-blocking mode, is the same logic that we mentioned to in the blocking mode. The main execption is with the assignment of the logic segements to different states within the scoped memory areas, this is essential in this mode, as these code segments are executed by different handlers as explained next. The code in Figure 6-24, shows the non-blocking version of the `ClientLogic` class; in the following, we present the main differences in this code, from the one defined for the blocking mode.

The first noticeable difference is the assignment of the maximum number of states in this logic; this value is related to the number of changes of the handlers executing this code. In this class, the `ClientLogic` class, this logic starts execution by the client calling thread, which has id responsible for the preparation and the sending of the packet to the server; once this packet is sent, the control over this logic is transferred to the communicator component, which receives this logic as an attachment with the registsered communication channel, which is registered for the READ readiness operation with the Selector. Once bytes received on this channel, the enclosing communiocator of this selector fires a free handler to continue the execution of the next segement of this attached logic, i.e. the `ClientLogic` class; this next segment is responsible of receiving the reply packet from the server, and processes it. Then, the process is repeated in the next periods.

```
public void runUpWard(int curLevel, final IComponent parentComponent) {
  synchronized (this) {
   if (curLevel == 0) {
    if (communicator == null) {
     communicator =
((CommunicatorCls)(parentComponent.getContainer().getComponents().get(
         "Communicator"))); //get the communicator
     channel= communicator.makeConnection(remoteAdress, 2190, 25);//connect
    }
    if (currentState() == 0) { }
    if (currentState() == 1) { }
   }
   if (curLevel == 1) {
    if (currentState() == 0) {
     writeOutputPacket();
     try {
      communicator.registerationQueue.add(getChannel(),
SelectionKey.OP_READ,this); //add this channel to the registseration queue
      communicator.theControllerChannel.openSignalFD(10); //wake up
     } catch (Exception e) {
      System.out.println("Exception....." + e);
     }
    } //end state 0

    if (currentState() == 1) { }
   }  //end level 1

   if (curLevel == 2) {
    if (currentState() == 0) { }
    if (currentState() == 1) {
     System.out.println("The Packet Number " + packetNumber++ + "Has been
written");     //The code displays messages to the user
     System.out.println("The client started to wait At" + tstart);
    //any other intermediate operation can be done here
    } //end state 1
   }  //end level 2
  }   //end runUpward
 }
 public void runDownWard(int curLevel, IComponent parentComponent) {
  synchronized (this) {
   if (curLevel == 2) { }
   if (curLevel == 1) {
    if (currentState() == 0) { }
    if (currentState() == 1) {
     readInputPacket(); //read the input packet
    }
   }
   if (curLevel == 0) {
    //we may close channel, reset values, ...etc,
    n = 0;    nn = 0;
    if (currentState() == 0) { }
    if (currentState() == 1) {
     SetState(0); //we reset the state machine
    }
   }
  }
 }
```

Figure 6-24 ClientLogic [Non-Blocking]

From this scenario, we can see that the logic has to be divided into two different segments. To implement this scenario, we assign a single state for each handler to execute its code within it. Hence, there are two states in this class; state (0) in which the calling thread starts to prepare and send the packet and printin logging statements, and step (1) in which the read-handler, which is released by the communicator, receives the incoming packet, and may process and decode it. One important additional step required in this mode, is the resetting of the state machine, when the handler finishes the depropagation from the stack and leaves level (0), this step is required to enable the state machine to start from the zero state in the next periodic cycle.

E- Read Logic, Write Logic and Connect Logic Classes

Each one of these classes define the logic of a corresponding network communication event handler, where at the client side, only the read, write, and connect are the only possible events. In this simple example, as we assume that the client uses the communicator component directly; then, we assume that the logic of any of these event handlers can be just used to monitor the occurenece of any of the network events. According to this, we have two possibilities; the first one is that each one of these classes can be simply a class that implements the `Runnable` interface, where the `run()` method of this class should define required the logic, an example of such a class is shown next.

```
Public void ReadLogic implements Runnable
public void run() {
//write a message, log the event, or record the time, ….
  System.out.println(“reading data ”);
 }
```

The other possibility is to define a class extending the `ISTackLogic`, this can be particularly important in the case that the required logic needs to create temporary objects; in this case, these objects have to be to be created in a memory with short lifetime, i.e. not in the level (0), the container memory area, nor in level (1), the component memory area, but in the level (2) or above of theSMA. The following code shows an example of such possibility.

```
public class ReadLogic implements IStackLogic
{ public void runUpWard(int curLevel, IComponent parentComponent) {
   if (curLevel == 0) {   }  if (curLevel == 1) {   }
   if (curLevel == 2) {
             System.out.println("Reading event on"+parentComponent);
  } } }
```

In the above example, the printing systemant involves the creation of a temporary object, which is created implicitly to hold the string resulting from joining the literal sring and the object interface reference. So, to ensure that it does not cause any memory leak, this statement is executed in level (2) of the memory.

6.5.2 The Server Side

The structure of the server side has a lot of similarities with the structure of the client side. Hence, the classes and logic used at the server side are close to those used at the client side.

The server side container contains a server component and a communicator component. So, the definition of these components and their properties is made in the `ServerSide` class, which is the container class at the server side; this is made in the overridden `BuildComponents()` method in that class. The basic operations of the `BuildComponents()` method are shown next.

```
  Callee server = new Callee();
  server.init(theCMA, 100000, LTMemory.class, isizesServer, msizesServer,
      generalFork.class); //initiate the server component
  server.setComName("Server"); //give the server component its name
  addComponent(server);   //add the server component to   this container
  ReusableRunnableStack task =
      server.addSMATask("ServerExecutor", RealtimeThread.class, null,
null, 10000, 10000, LTMemory.class, null,
          ServerLogic.class); //create a new SMA task
  task.setParameters(LTMemory.class, 3, new LTMemory[] {theContMA,
theCMA}, initM, maxM,
      server); //assign the parametrs of the RRS of the SMA task
  CommunicatorCls communicator = new CommunicatorCls();
  communicator.init(CMA, 10000000, LTMemory.class, isizesComm,
 msizesComm,
DualFork.class); //initialize the communicator
  communicator.setComName("Communicator");  //Give the name
  addComponent(communicator);                //add to the container

  ReusableRunnableStack serverLogic =
      communicator.addSMATask("ServerCommunicator", RealtimeThread.class,
null, null, 10000, 10000, LTMemory.class,null,
          CommunicatorAsServerLogic.class); //create a new SMA task
  serverLogic.setParameters(LTMemory.class, 3, new LTMemory[] {theContMA,
theCMA}, initc, maxc,
      communicator); //assign parameters for the RRS of the added SMA
task in the communicatoe
  communicator.createHandlersPool(20, 120, 120,20); //creates pools
  communicator.setReadLogic(ReadLogic.class);
  communicator.setWriteLogic(WriteLogic.class);
  communicator.setAceptLogic(AcceptLogic.class);
  communicator.createObserver(0);  // the server side mode
```

Due to the similarity in the architecture, the `BuildMethod()` of the `ServerSide` class is very similar to the same method in the `ClientSide` class, with few exceptions. The `BuildMethod()` method has the following operations:

- Creating a server component, initializing it, setting a name for it, and finally adding it to the container.
- Creating a task to run within the server component, where this task is a real-time thread with a reusable runnable stack of three levels of scoped memories, the container memory level, the component memory level, a temporary scoped memory above them. This reusable runnable stack runs the logic defined in the `ServerLogic` class.
- Creating a communicator, initializing it, giving a name to it, and adding it to the container.
- Creating another task of a real-time thread, initialize it, and add it to the Communicator component, where the logic executed by this reusable runnable stack is defined in the `CommunicatorAsServerLogic` class.
- Configuring the sizes of handlers' pools of the communicator component, and assigning the classes `AcceptLogic`, `ReadLogic`, `WriteLogic` classes, which define the logic to be executed by the handlers of the networking communication events.
- Setting the communicator component to run in the server mode.

In the following, we will discuss the structure of the classes used to build the structure and the logic at the server side.

A- CommunicatorCls

The same class used at the client side, but configured to run in the server mode.

B- CommunicatorAsServerLogic

This class is identical to the one used at the client side, with a simple change; this change is the calling of `runAsServer()` method of the parent component, the communicator, instead of calling the `runAsClient()` method.

C- Read Logic, Write Logic and Classes

In our example, the `ReadLogic` and `WriteLogic` classes are the same ones that are used at the client side, as in this example each one of them prints a message when its corresponding event occurs.

D- The Accept Logic

The `AcceptLogic` class, in both the blocking and the non-blocking modes in our example, defines the logic of the processing of the accept event when a new connection is received. In the following, we present the structure of this classs in both of these two modes**.**

- **The Blocking Mode**

The `AcceptLogic`, in this example, is responsible of sending a reply packet to the client. The implementation of this class has many similarities with the `ClientLogic` class, as both, in the emlated blocking mode, are extending the `NWHandlerStackLogic` class. As it was the case in the `ClientLogic`, this class defines two buffers; one for the input buffer, and the other for the output buffer, in addition to a reference to the communicator component, in which it is running.

The logic of this class is executed by the acceptor handler, which is is released from within the communicator component, once the connection to this component is made. The acceptor handler runs, as the client logic, in a memory stack of three memory levels, the container memory area, the communicator component memory area, and finally in the handler's own temporary scoped memory area. In the following, we present the execution sequence of the logic, shown in Figure 6-25, within these three memory areas.

- **IF[UpWard], i.e. propagating upwards**

⇒ If [curLevel==0], i.e. entering the ContainerMA:

In this level, a reference of the Communicator component is retrieved, only in the first cycle, as it was the case in the `ClientLogic` class.

⇒ If [curLevel==1], i.e. entering the CMA:

Again, like the `ClientLogic` class, the first part of the functionality of the acceptor handler is executed here, where it involves the following steps:

- Get the channel assigned for the communication with the connecting client.
- Add this channel, with the current handler as an attachemnt, to the registeration queue, to be observed for the read-readiness event, i.e. arrival for the input packet bytes.
- Activates, the selector to register all the saved elements in the registeration queue.
- Waits For the arrival, of the incoming packet bytes; then start to read these bytes, once they arrive to the communicator component, which notifies the acceptor handler to process this operation. This is repeated as long as there are remaining bytes of the incoming packet.

⇒ If [curLevel==2], i.e. entering the temporary scoped MA:

As this memory, as explained before for the `ClientLogic` class, is more approporiate for the operations that involve creation of hidden and short time objects; then, the operations of printing logging statements, and/or decode the incoming packet, and do further processing on it can be done here.

```
public void runUpWard(int curLevel, IComponent parentComponent) {
  if (curLevel == 0) { //runs in ContMA
   if (communicator == null) {
    communicator =
((CommunicatorCls)(parentComponent.getContainer().getComponents().get(
        "Communicator"))); //get ref to the communicator    }
  }//end level 0
  if (curLevel == 1) { //runs in CMA
   try {
    SelectionKey key = ((EncapsulatedHandler)getHandler()).getSelectionKey();
final SocketChannel channel = (SocketChannel)key.channel(); //get the channel
    communicator.registerationQueue.add(channel, SelectionKey.OP_READ,
        getHandler()); //register the channel
communicator.theControllerChannel.sigQueueToSignalFD(12, 10);//wake up selector
inbuf.clear();
    do {
     synchronized (getHandler()) {
      getHandler().wait();
     }                           //end synchronized
     try {
      nn += channel.read(inbuf); //read into the buffer
     } catch (Exception w) { }
    } while (nn < 40);
    inbuf.flip();
   } catch (Exception r) {
    Syustem.out.println("Exception--> " + r);
   }
  }                    //end level 1
  if (curLevel == 2) { //runs in temporary scoped memory area
   System.out.println("The Packet [" + ++PacketCount + "] has been received");
   //decode and process the received packet [may create objects]
   //.................
   }
  }                      //end level 2
 }                       //end runUpWard

 public void runDownWard(int curLevel, IComponent parentComponent) {
  if (curLevel == 2) { } //in temporary scoped memory area
  if (curLevel == 1) {   //in CMA
   outBuf.clear();
   for (int i = 0; i < 250; i++) {
    outBuf.putInt(mm.nextInt(250)); //write to the buffer
   }
        //send the packet
   outBuf.flip();
   try {
    channel.write(outBuf); //write to the channel
   } catch (Exception e) {
    System.out.println("*-----Exception----*" + e);
   }
  }

  if (curLevel == 0) { //in ContMA
   //reset the values, recycle any unrequired object ..etc.
   n = 0;  nn = 0;
  }
 }
```

Figure 6-25 AcceptLogic [Emulated-Blocking]

- **IF[DownWard], i.e. propagating downwards**

  ⇒ If [curLevel==2], i.e. leaving the temporary scoped MA:

  Again, this step is executed in the same memory area with the last step, so the code can be divided between them.

  ⇒ If [curLevel==1], i.e. leaving the CMA:

Here, as this is the CMA, the second part of the componwnt functionality is executed, i.e. the reply packet is prepared and sent back to the caller client over the assigned channel.

⇒ If [curLevel==0], i.e. leaving the ContainerMA:

Here, all the resetting operations, and preparation for the next cycle are made.

```java
public void runUpWard(int curLevel, IComponent parentComponent) {
 synchronized (this) {
  if (curLevel == 0) { //runs in ContTMA
   if (communicator == null) {
    communicator =
((CommunicatorCls)(parentComponent.getContainer().getComponents().get("Communicator"
)));//get the communicator from the container
   }//end if null
  } //end if level=0
  if (curLevel == 1) { //runs in CMA
   if (state == 1) { //executed by accept handler
    readInputPacket();//read incoming packet
   }
   if (state == 2) {  //do nothing
   }
  }

  if (curLevel == 2) { //runs in the temporary memory
   if (state == 1)     //executed by read handler
   {
    System.out.println("The packet number" + PacketCount++ + "has been received");
    DecodePacket(); //do some processing on the packet
   }                //end state 1

   if (state == 2) {//do nothing
   } //end state=2
  }  ////level 2
 }   //synchronized
}    //runUpward
public void runDownWard(int curLevel, IComponent parentComponent) {
 synchronized (this) {
  if (curLevel == 2) { //do nothing
  }
  if (curLevel == 1) { //runs in CMA
   if (state == 1) {//do nothing
   }
   if (state == 2) {
    writeOutputPacket(); //Write and send a reply  Packet
    communicator.registerationQueue.add(getChannel(), SelectionKey.OP_READ, this);
    communicator.theControllerChannel.sigQueueToSignalFD(12, 10);
   }                   //end state 2
  }                    //end level 1
  if (curLevel == 0) { //runs in ContTMA
   if (state == 1) {//do nothing
   }
   if (state == 2) {
    setState(0); // reset the state machine
   }
  }
 } //end synchronized
}
```

Figure 6-26 AcceptLogic [Non-Blocking]

- The Non-Blocking Mode

In the same way described above, the main difference in the implementation of this class in the non-blocking mode, from its implementation in the blocking mode, is in the division of the logic into different segments, where each segment is executed by one of the threads/handlers in exactly one state. In the logic defined in this class, see Figure 6-26, two states are defined, state (0), the idle state, state (1), the receive-and-reply state.

In the idle state, the logic component is in the pool, waiting to be released when a new connection is made, and no running handler is executing it. When the communicator receives a connection from the server, the acceptor releases an acceptor handler and assigns this logic to it, then it forwards the state machine to the next state, i.e. the receive-and-reply state.In this new state, the acceptor handler reads the incoming packet, decode and process it; then write a reply packet to the server; then finally it resets the state machine. As there is only one handler processing this logic, there is only one non-idle state.

It is important to note that in the implementation of the Stack Logic pattern in the non-blocing mode, e.g. in both the `ClientLogic` class, and `AcceptLogic` class, requires that the logic within the `runUpward()`, and `runDownWard()` methods have to be enclosed within a `synchronized(this)` statement, to avoid the race conditions, in order to ensutre that the handlers collaborating in the execution of the different segements of the logic in the different states of these functions do not overlap their execution.

6.6 Summary

Although RTSJ provides a lot of features to enhance the predictability of the programms built using the Java language, it has not provided communication mechanisms suitable for distributed real-time applications. So, the aim of the chapter was toward providing a concrete design model of a component using our proposed framework that integrates the new features added by the RTSJ with the most efficient communication mechanisms currently provided with the Java Language in order to have a real-time reconfigurable component that can be a base for building many real-time middleware solutions.

In order to design the required component, we analyzed first the basic strategies and models for I/O over networks. After that, we discussed how the current Java packages support these models. Then, in order to choose the best flexible model, we discussed the existing server design patterns, where, found that the non-blocking communication pattern, is the most efficient and flexible pattern for communication, and it can even be used to emulate several communication strategies, and it can be applied to both the server side and the client side, So we presented our design of the proposed remote communication component, which integrates the predictable RTSJ bsed component model provided in Chapter 5, with the non-blocking mechanisms provided by Java and discussed in this chapter, to build a real-time reconfigurable Communicator component, which can be used to build real-time middleware solutions, as the remote middleware modelpresented in the next chapter.

Finally, to illustrate how the communicator component can be integerated with the other parts of the framework, i.e. the memory model and its assocuiated patterns presented in Chapter 5, we provided at the end of this chapter a simple client-server example that enables the exchange of byte packets between the client and the server. In the next chapter, we extend this example to see how we can support remote method invocatio

Chapter 7

Reconfigurable Real-time Middleware Model

Java’s Remote Method Invocation; RMI, is the basic communication middleware model provided in the Java language, to take the error-prone low level communication with sockets and streams out of the programmer’s hand, in order to enable accessing remote objects residing in another JVM, in a similar way to accessing local objects within the same JVM. Moreover, these two JVMs that hold the caller and the called objects may reside on the same machine, or they may reside in two different machines, which are linked together by a network.

In this chapter, we build our own real-time middleware model using component framework presented in Chapter 4, and the remote communication component presented in Chapter 5. Before presenting our proposed model, we discuss the fundamentals of the remote communication process, to clarify the basic general patterns of it; then, we present an overview of the RMI-HRT package, which we modified support the non-blocking pattern required for our model, where this package is an open source package of a real-time architecture model provided in (Tejera, Alonso et al. 2007). Also, we give an overview of the serialization/deserialization process used for transferring Java objects over the network between the JVMs holding the communicating objects.

Then, we show how we integrate the remote communication component within the architecture of the RMI-HRT, to build our new proposed middleware model, to provide a new enhanced real-time middleware model of the RMI. We present the modified architecture of the model at both the server side and the client side, to show how we can use the components at both the client side and the server side, and how they integrate with the communicator sub-component within the forked memory model of the component model, to provide the basic elements of the RMI architecture. Then we discuss how the RMI protocol should be changed at both the server and the client side in our model, to support the different execution modes offered by the communicator component. Also, we analyze the difficulties and constraints of implementing the proposed model in the RTSJ that result from the constraints and the properties of its memory and scheduling models, and we evaluate how the memory management patterns presented in the component model both at the server and the client side can be integrated with the other patterns of the middleware, to enhance its functionality and to support the future call pattern of executing calls.

7.1 Basic Patterns of Remote Communication

Remote communication is the operation in which an object on a certain machine can invoke a method of another object from within another machine. The Broker software pattern, first described in (Voelter, Kircher et al. 2004) is the principal pattern for building remote communication. The Broker pattern aims to hide the complexities of using the networking communications and the integration of the heterogeneous component into coherent application, so that the developer can keep focus on solving the application problems.

The Broker pattern is applied on both sides of the remote communication operation; the client side and the server side, so that the client can exchange the requests and responses with the remote object. The Broker is a compound pattern which itself consists mainly, as shown in Figure 7-1, of the following basic sub-patterns (Voelter, Kircher et al. 2004):

- **Requestor**; this is at the client-side to construct and make remote invocations to the server over the network on behalf of the client.
- **Invoker**; this is at the server-side and it is responsible for invoking locally on the server the requested operations of the remote object on behalf of the client.
- **Marshaller**; this is responsible for transformation of requests' parameters and responses from programming language defined data types into raw bytes that can be sent over the network stream and vice versa, i.e. it performs the demarshalling operation by rebuilding the original parameters and/or return parameters out of the received bytes from the network. In the Remote communication middlware model, two Marshallers have to exist, one at the client side to convert the request parameters into raw bytes and send them over the network to the server, and it reconstructs the request parameters from the received bytes representing the return value when they arrive to the client from the server side. On the server side, another marshaller exists to do the opposite operations, i.e. receive the raw bytes and use them to reconstruct the request parameters, and convert the return parameter into raw bytes to be sent to the client.

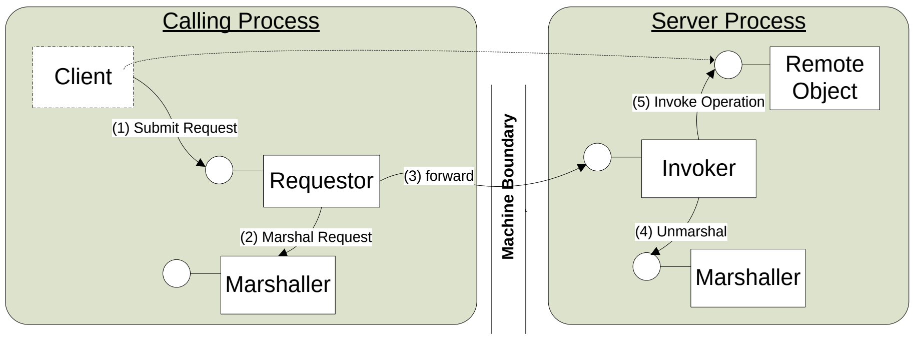


Figure 7-1 Broker Pattern Architecture

In addition to the above basic patterns, the following patterns integrate with them to provide the basic remote communication patterns, see Figure 7-2.

- **A Client Proxy**; this is a local pattern within the client process that offers the same interface as that of the remote object in order to make the remote operation invocations look like local invocations.
- **An Interface Description;** this is used to make the interface of the remote object known to the clients, and it is used to build the Client Proxy of that remote object within the client.
- **The Client Request Handler and the Server Request Handler;** these patterns form a layer beneath the Requestor and the Invoker to handle sending, receiving, and dispatching of requests.
- **Remote Communication Errors;** this pattern is responsible for forwarding the distribution errors, e.g. network failure, invalidity of the server, etc. to the client.
- **Look Up Pattern;** clients within the application need to get a reference to the remote object in order to call its methods, where the reference to the remote object must identify both the address of the machine on which it exists, and the *Object ID* of the remote object within the remote server application. The Look Up pattern is responsible of getting the actual address of a remote reference, where this remote reference can be statically hold within the client machine or dynamically retrieved from a central repository before making the remote calls, where in case of the dynamic model, the server object has to register itself within the central repository. Accessing the RMI Registry is an example of this pattern, where the registry represents a central repository of the remote references, that registered within by the remote object and clients have to look up within it for the required remote reference.

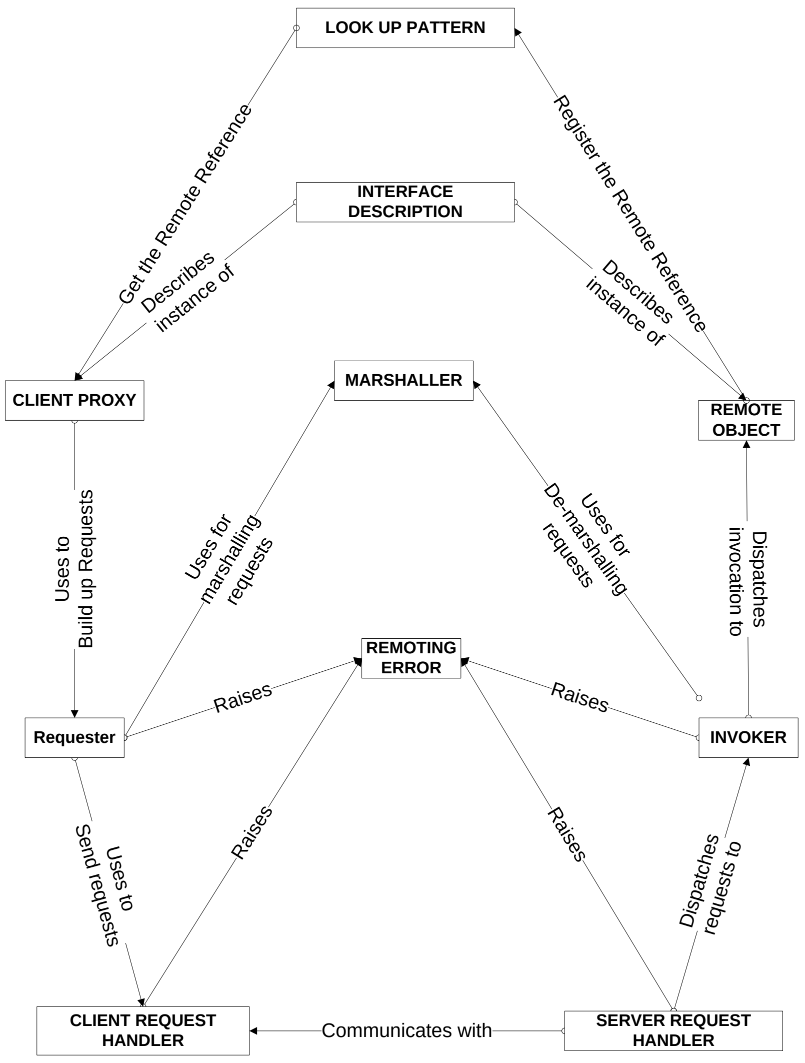


Figure 7-2 Basic patterns of the remote communication

7.2 Extension Patterns of Remote Communication

In addition to the basic patterns mentioned above, some extension patterns can be integrated with the basic patterns in order to provide some specific functionality. Examples of these patterns include (Markus Völter, Kircher et al. 2004):

- **Invocation Interceptors Patterns;** These patterns can be used in case of the requirement of adding security credentials to the remote invocation where the invocation interceptors can intercept the invoked call at the client side before sending it to the server to add the security credentials, and they are added as well at the server side to check for these credentials before invoking the method.
- **QoS Observer Patterns;** this pattern is used in distributed real-time systems that need to monitor or measure the performance of various parts of the remote communication model, e.g. the Server Request Handlers, the remote object, etc.

- **Location Forwarders (Markus Völter, Kircher et al. 2004);** One example of the use of this pattern is in Load balancing, where it can be used to forward invocations between several server applications transparently.
- **Life Cycle Management Patterns;** from the life cycle point of view, there are three basic models of the server object:

1- **Static Instances;** the life time of the server object is the same as the life time of the server application.
2- **Per-Request Instances;** the life time of the server object is for the length of the request processing, i.e. the server application creates the server object when the request is received, and destroys it once the request is finished. This model is used in highly concurrent environments to save the resources as in many of distributed real-time systems. Both the Lazy Acquisition Pattern (Kircher 2001) and the Pooling Pattern (Kircher and Jain 2002) can be used for managing the life cycle of this model. In the case of the Pooling pattern, the pattern manages a pool of reusable remote object instances where the object is retrieved from the pool when the request is received and it returns back to the pool when the execution of the request ends. On the other hand, the Lazy acquisition pattern loads and executes the server object only when it is needed otherwise it never loads it.
3- **Client-Dependent Instances;** the client remotely creates these instances on the server. The Leasing pattern (Markus Völter, Kircher et al. 2004) can manage the lifecycle of this model, as this pattern deactivates the remote object after a pre-defined period of time, if the client has not renewed the leasing period assigned to it.

7.2.1 Invocations Patterns

Invocation patterns are a set of patterns that can be used in the Remote middleware to specify the way in which the client makes the invocations; in other words it defines how the client will behave to start the invocation and how it will behave during the execution of the remote method at the server, and finally how it would get the result of the execution from the server, if there is an expected result to come back from the server. In the following we define the basic invocation patterns commonly used in the Remote middleware:

- **Synchronous Blocking Invocation;** in patterns of this category, as shown in Figure 7-3, the client sends the request through the Requestor to the server process; in the server process, the invoker receives the call-request and invokes the required call from the remote object and waits for the result; once the method finishes its execution, the result is sent back to the client side through the invoker. During these operations the calling thread at the client side is blocked waiting for the result of the call execution. Hence, it cannot be reused for any other processing while waiting for I/O to complete. This causes a waste of the client resources when multiple calls are to be executed on remote object(s) concurrently, especially for long duration method calls, as this will cause multiple threads to be blocked without ability for reuse. Hence, the use of this mechanism is highly constrained for many real time systems, especially those with limited physical resources.

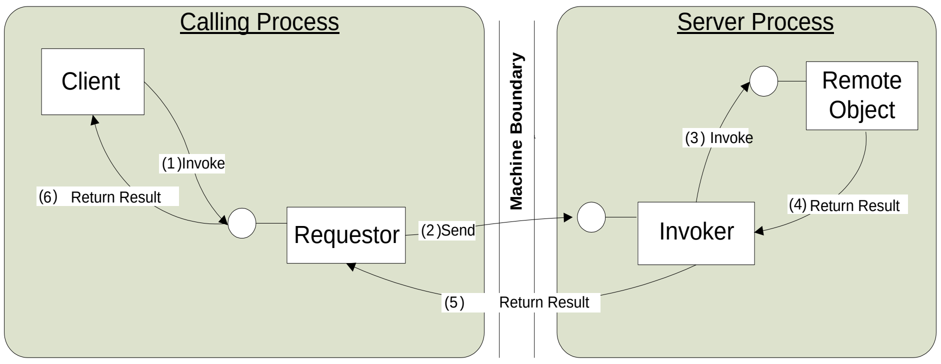


Figure 7-3 Synchronous Invocation Pattern

- Asynchronous Fire and Forget I/O; this is the simplest form of communication, where, as shown in Figure 7-4, the caller thread sends a request to the server and returns immediately to continue its execution without waiting for any return result or acknowledgement from the remote server thread. Hence, it has no reliable or predictable behavior, as the caller will not know if the call has been executed or not, as it may be lost before arriving to the server. The requestor objects, a proxy object responsible of managing the call execution on behalf of the client, can handle the client request in various ways as follows;

1- It can execute within the client thread context hence, an operating system asynchronous support is necessary to avoid blocking the client thread.
2- The requestor cans spawn a new dedicated thread to invoke the remote operation. This is suitable when the number of concurrent calls imitated by the client is few; otherwise, there will be a noticeable concurrency overhead due to the lock contention and race conditions, etc.
3- As the pattern itself is not reliable, the requestor can execute the call using one of the unreliable protocols (e.g. UDP) which is faster and has less resource requirements.

This pattern can be helpful in implementing a limited set of event driven systems as a mechanism for creating remote operations that need not to be reliable (e.g. nonsensetive logging systems). However, due to its unreliability, this pattern cannot be used in implementing sensitive operations especially in real-time systems, e.g. updating sensors measurements in patients monitoring systems.

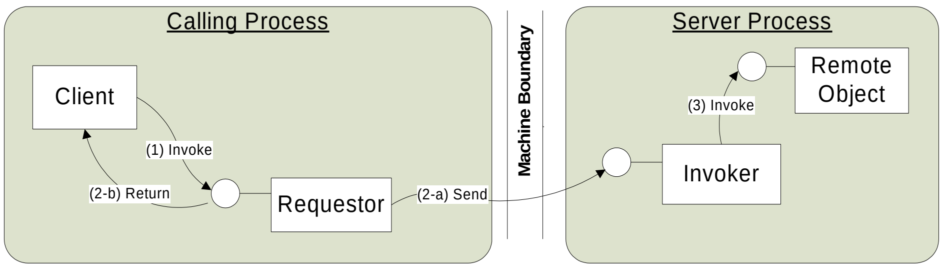


Figure 7-4 Asynchronous Fire and Forget Pattern

- **Sync with Server Pattern:** This pattern is an extension of the fire and forget pattern to try to enhance its reliability. In this pattern, shown in Figure 7-5, as in the fire and forget pattern, the client sends the call to the server through the requestor. However, instead of continuing execution immediately after delivering it to the requestor, the client thread waits to get an arrival acknowledgement from the server side that the call has been received and it is going to be processed. Once the client has received this acknowledgement, it continues its execution asynchronously.

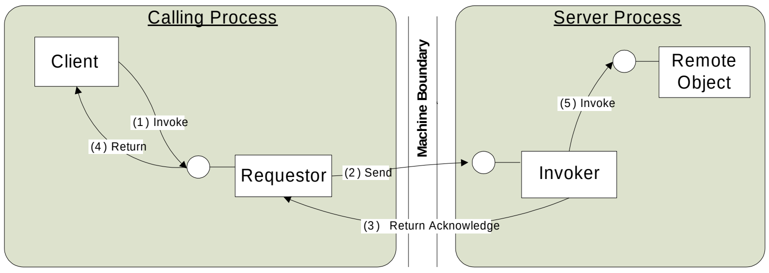


Figure 7-5 Synch with Server Pattern

The implementation of this pattern requires the invoker object, the object that receives the request and executes it on the remote object at server side to be able to send the reply asynchronously either truly using operating system facilities, or by emulation by spawning a new thread to process the request while retraining the acknowledgement to the client.

Due to its enhanced features, this pattern can be seen more efficiently in event-triggered applications where the client side generates events to be delivered to remote servers to take corresponding reactions to them while the client needs only to ensure the delivery of the event remotely without a need to wait to be informed by the server of any reaction or reply. Due its reliability in delivering the request to the server, this pattern is more reliable to be used in real-time systems (e.g. sensors send to monitor unit instant readings).

- Poll Object Pattern; in contrary to the fire and forget pattern and sync with server pattern, in which the client needs no result of the remote request to return from the server, the poll object pattern is to be used when the client depends on the result of the invocation for further computation, while these results are not needed immediately. Hence, the client can continue execution or do some useful computation instead of blocking idle, and check for the results availability later when needed.

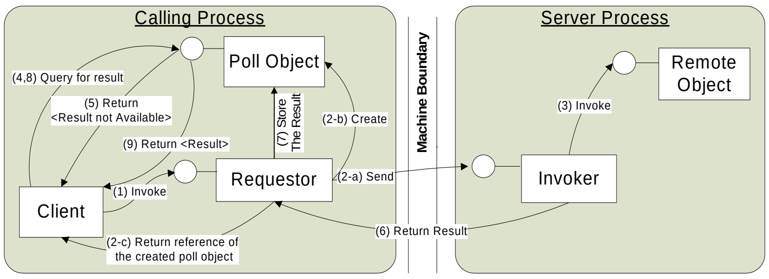


Figure 7-6 Poll Object Pattern

The scenario of operation of this pattern is shown in Figure 7-6, where the client sends the request to the requestor object, which in turn creates poll object to store the returned result of the remote execution, and controls returns immediately to the client thread with a reference to this poll object hence, the client can continue execution asynchronously. At the same time, the requestor spawns a new internal slave thread to make a synchronous call of the method on behalf of the client; this slave thread is responsible for waiting for the returning result of the call and storing it when it arrives in the corresponding poll object. When result is needed, the client calls on the pool object to check the availability of the result, so it can retrieve it from the poll object. Otherwise, in case of non-availability, it can choose either to continue asynchronous execution to do useful processing or to block waiting for the result to arrive.

- Result Callback Pattern; as the poll object pattern is using a synchronous execution model internaly on behalf of the client, it is more useful when the time until the result become available can be estimated and it is short, but enough to do useful work at the client side. Otherwise, to avoid waiting for longer operations, the client needs to be informed immediately when the results are available to the requestor. In the result callback pattern, an event driven approach is adopted for the internal design to immediately inform the client.

In this pattern, as shown in Figure 7-7, the client instantiates a callback object and passes it with the invoked operation to the requestor and returns immediately to continue execution. The requestor in turn, sends the invocation to the server. Once the server finishes the execution, a predefined callback method is called on the callback object passing it the result of the remote invocation.

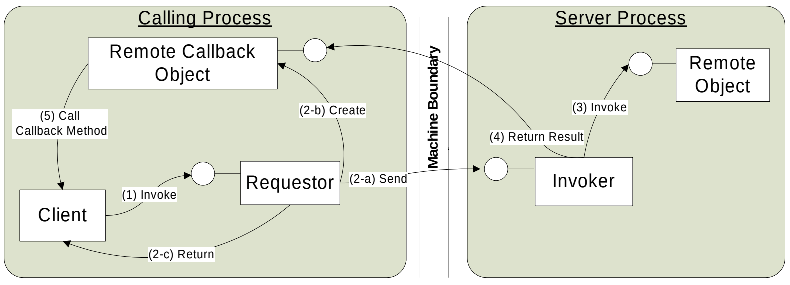


Figure 7-7 Result Callback Pattern

This callback method can interrupt the operation of the client thread to notify it to react to the completion of the invoked operation. According to the implementation of the calling systems, the calling of the callback method that notifies the client can be done in one of two ways:

1- Locally at the client side, when the client synchronously invoke the remote method using a separate thread, and once the result is returned back to this local thread, it notifies the client using the provided callback object.
Remotely from the server side, when the callback object is implemented as a remote object, in order to allow the callback method to be called remotely from the server.

7.3 RMI-HRT Overview

In order to develop our proposed real-time remote communication pattern, we based our model on the classes provided by the open-source modules, that implement the HRT-RMI model presented in (Tejera, Alonso et al. 2007). The HRT-RMI was built as part of the High Integrity Java project in order to provide a real-time model of the Java RMI using the new features presented by the RTSJ earlier in this thesis. The design of the HRT-RMI model was built as a modification to the basic RMI model (Sun Microsystems Inc 2004), where it shares many of the basic patterns used in the Java RMI, while its implementation has been included some modifications to the RMI Patterns by modifying the classes that implement the Java RMI in the Jamaica JVM by using some of the RTSJ features in order to support the requirements assumed in the HRT-RMI model.

In this section we present the general hierarchy of this open-source model, in order to clarify the changes that we made to it to build our own model, which is presented later in this chapter, where in our presentation of the HRT-RMI middleware Model, we will show our view of the basic parts of the model by mapping these parts it to their corresponding general remote communication patterns presented earlier in this chapter, with an explanation of the algorithms done in each individual pattern and the Java classes defined by the authors of the model in order to build each pattern.

7.3.1 The Broker Pattern

The central pattern in the remote communication middleware is the Broker pattern, which itself is composed of a set of patterns that exist on both the server side and the client side. In the following we will show the classes and the operations of each of these patterns, see Figure 7-8.

A- Client Side

The broker pattern of the client side in the RMI-HRT has only a few changes from the original model of the Java RMI as explained in the following:

- **The Client Proxy;** at the client side the calling thread has to have a proxy that has an interface that offers the same methods offered by the remote server, in order to hide the details of the remote calling over the network. At the client-side of the HRT-RMI model, this proxy is called the `HrtStub` and it is created as an instance of the class that has a name of the pattern `[RemoteObjectName]Imp_HrtStub`, where the `RemoteObjectName` is the class name of the remote object. The `[RemoteObjectName]Impl` class is created using a modified version of the Java's rmic tool that is used by the Java RMI, where the modifications done on this tool are basically responsible of creating both the `[RemoteObjectName]Imp_HrtStub` class, and the `[RemoteObjectName]Imp_HrtSkel` class, in order to add the set of the features required to have the a real-time predictability as specified in the HRT_RMI model as explained in (Tejera, Alonso et al. 2007).
- **The Client Requester Handler;** in the HRT-RMI model, this pattern is represented partially by the `UnicastHrtRef` which is responsible for building the client session with the server through the method `createClientReservation()`, see Figure 7-9, which includes the following handshaking operations:

1. Send the Client Identifying name.
2. Send request header.
3. Waits to receive request-arrival acknowledge from the server-side.

The invocation of the method itself has been moved in the HRT-RMI model, to be directly made from the Requestor.

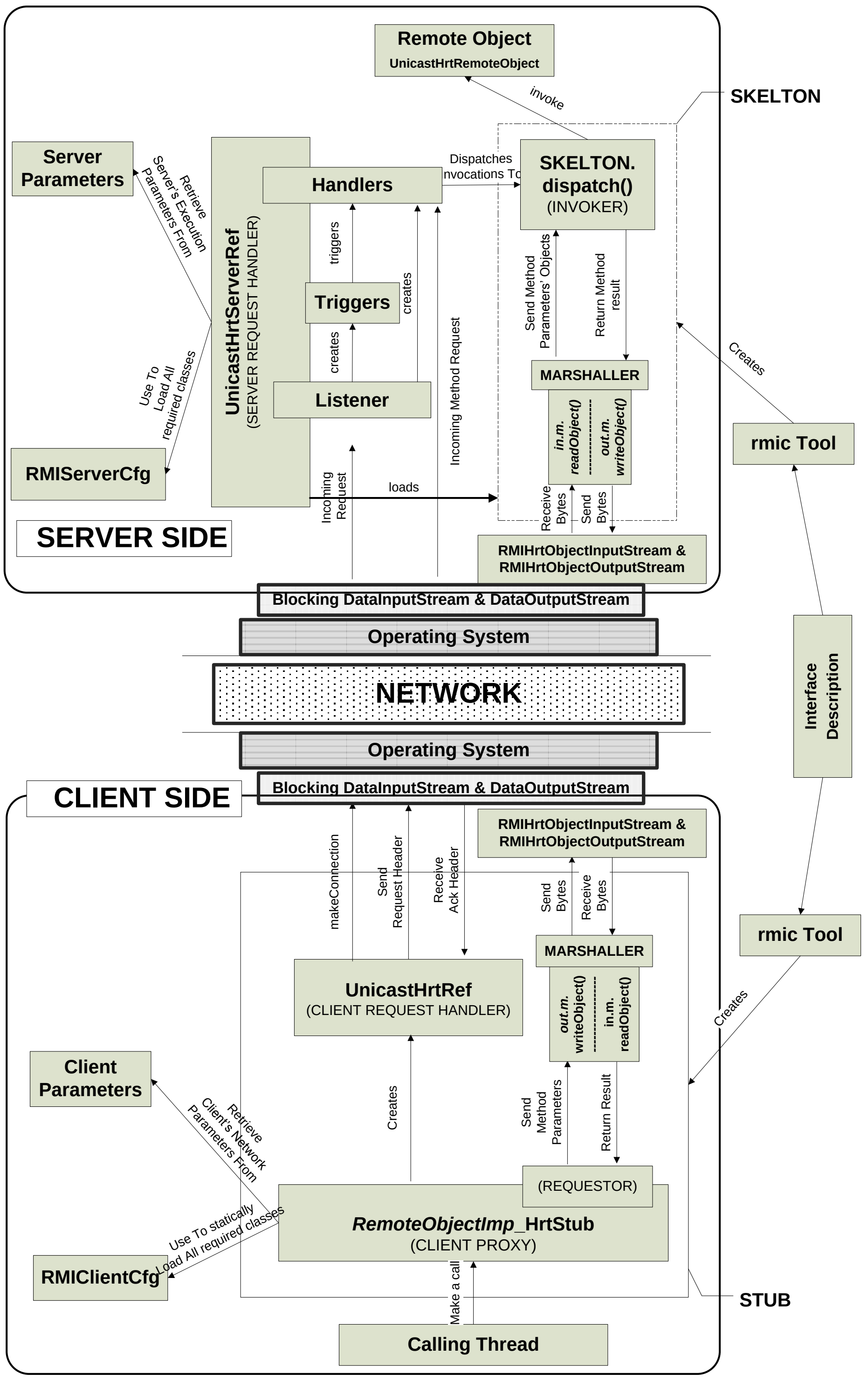


Figure 7-8 Hierarchy of the Broker Pattern of the HRT-RMI open source model

- **The Requester;** the requestor pattern is the internal pattern within the proxy pattern, and it provides the implementation of the remote methods at the client side, where this implementation receives locally the call of the remote method and performs it remotely over the network using the marshaller pattern, and it waits to receive the results of the call in order to return it back to the calling thread.

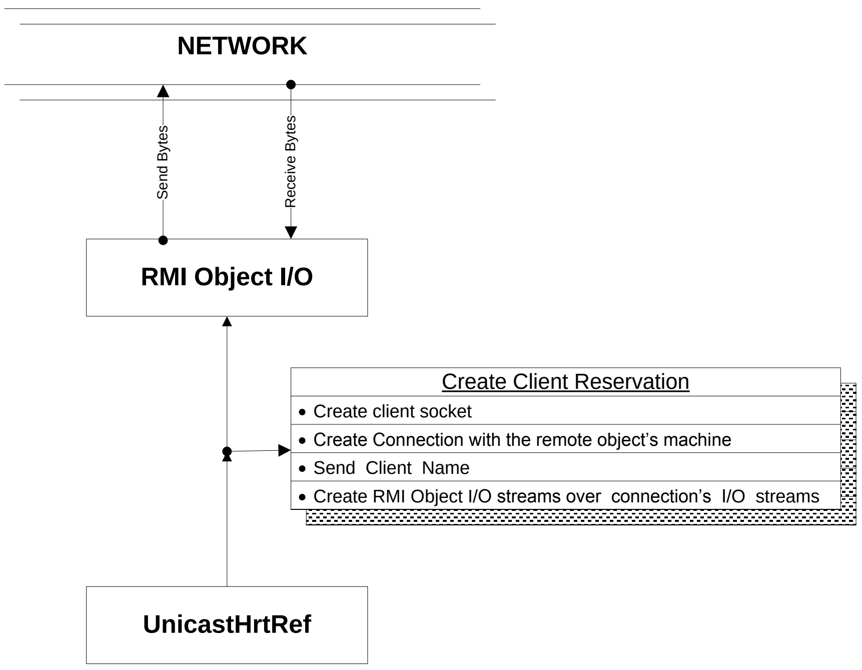

NETWORK
Send Bytes
Receive Bytes
RMI Object I/O
Create Client Reservation
• Create client socket
• Create Connection with the remote object's machine
• Send Client Name
• Create RMI Object I/O streams over connection's I/O streams
UnicastHrtRef

Figure 7-9 Operations of the createClientReservation() Method

- The Invoker; as it was in the Java RMI, The skeleton is representing the Invocation Pattern in the HRT-RMI model, where the skeleton receives the call request in the form of a message forwarded to it by the Call Handler Thread through the call of the `incomingMessageCall()` method of the skeleton object, then it invokes the required method through the method `dispatch()`.

B- Server Side

Most of the changes provided in the RMI-HRT were in the server model in order to enhance its predictability, as will be explained in the following:

- The Server Requester Handler Pattern; the main functionality of this pattern is encapsulated in the `UnicastHrtServerRef` class that extends the `UnicasHrtServerObject` class and it defines two main methods as follows, see Figure 7-10.

`exportObject()`; this method called from the RemoteObject either implicitly when it is created if the Remote Object is extending the `UnicastHrtRemoteObject`, or explicitly if the `RemoteObject` is created by extending the `HrtRemote` interface. This method is responsible for doing all the initialization required for making the server-side ready to handle all the incoming requests to the remote Object. These initialization operations include:

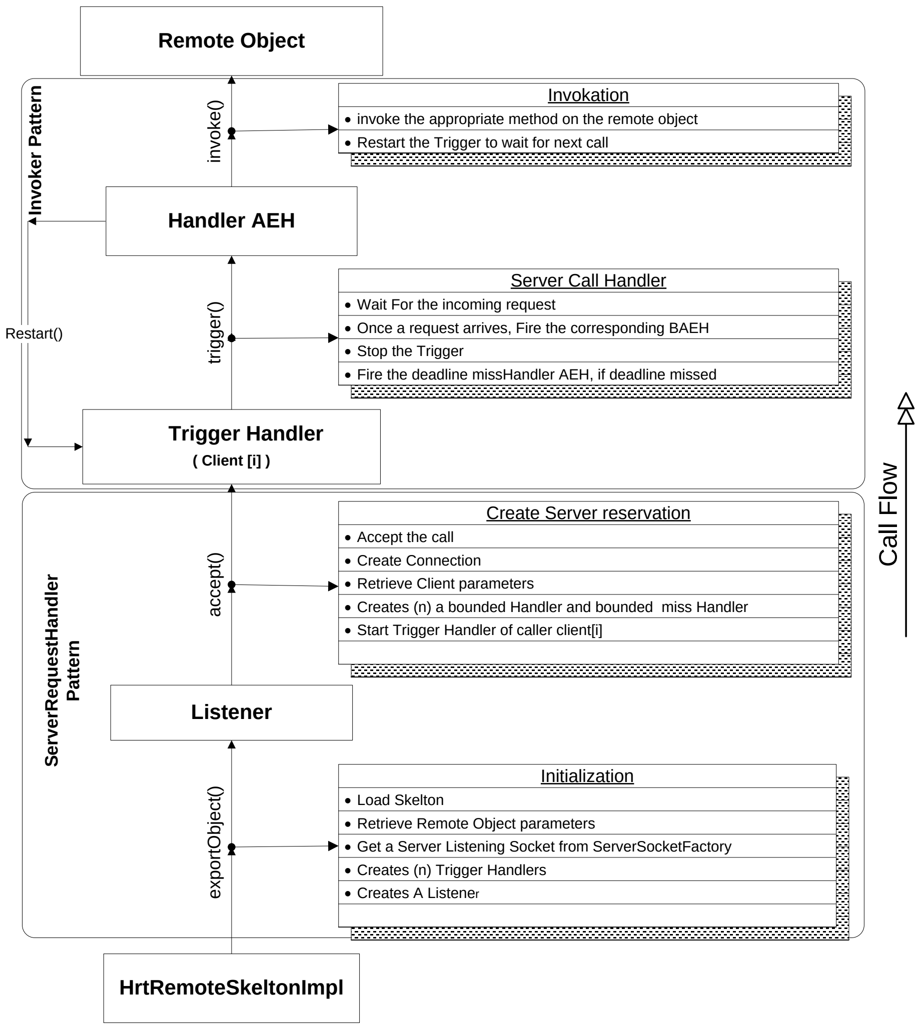

Remote Object
Invoker Pattern
invoke()
Invokation
• invoke the appropriate method on the remote object
• Restart the Trigger to wait for next call
Handler AEH
Restart()
trigger()
Server Call Handler
• Wait For the incoming request
• Once a request arrives, Fire the corresponding BAEH
• Stop the Trigger
• Fire the deadline missHandler AEH, if deadline missed
Trigger Handler
(Client [i])
Call Flow
ServerRequestHandler Pattern
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]
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 7-10 Call Handling at the Server Side

1- **Loading the Skeleton object;** the skeleton object in the HRT-RMI model is created *statically* by a modified version of the rmic tool; i.e. same as in Java 1.1, and not dynamically by reflection as in Java 1.4+. Hence, this method loads this remote object's skeleton from the disk to be ready for handling the requests.

2- **Creating a Listener**; the Listener is created as an instance of the `NoHeapRealtimeThread` class and it is responsible for calling the `createServerReservation()` method which is explained next.

3- **Retrieving the local remote object parameters;** the internal structure of the server request handler includes a set of real-time threads and Asynchronous Event Handlers which has to be assigned sporadic parameters in order to configure them, as will be explained later in this chapter. So, this method loads those parameters.

- `createServerReservation();` this method is called from within the Listener's realtime thread, in order to initialize a session between the server and the client. The server can have a maximum of *n=maximum number of Clients* registered with it and the `createServerReservation()` method is responsible of building sessions with all these registered clients, so it is called as many times as the number of the clients registered with the server, where it is responsible of doing the following for each client,

4- **Creates a connection object of the class** `UnicastHrtRMIConnection`**;** the connection object is using the Java data stream model, i.e. the Blocking I/O model with an associated `DataInputStream` object for reading from the network stream and another object of the `DataOutputStream` type to write to the output stream.

5- **Retrieve the Client Parameters from the server machine;** it retrieves all the clients' network parameters and real-time parameters that are statically registered within the class file `ServerCfgCls`.

6- **Creates a Runnable Object for the Call Handler;** the logic defined by this `Runnable` will be used as the logic for all the call handlers. The basic operation of this logic is to check the type of the incoming message from the client, and either reply to it directly with an acknowledgement if it was just a pinging message; or in the case of a message call, the handler forwards the call to the Invoker through the `incomingMessageCall()` method to react to this event.

7- **Creates set of *n* Trigger Handlers Objects**; where each trigger is created as a real-time thread assigned parameters loaded with the client parameters. The trigger handler is responsible for waiting for the incoming messages from a certain client in order to fire the event controlling the caller handler associated with the invocation coming from this client, and then return back to wait for other messages coming from the client.

8- **Creates a set of *n* Bounded Asynchronous Event Handlers as Call Handlers;** where this call handler is assigned server centric sporadic parameters which are assigned statically within Java classes that are loaded during initialization of the remote object as stated before.

9- **Creates a set of *n* Bounded Asynchronous Event Handlers as MissDeadLine Handlers;** these handlers are created to be executed if the call handler of the request coming from this client misses its deadline. Again, these handlers are created with sporadic real-time parameters loaded as part of the clients' parameters.

7.3.2 Configuration and Parameter Assignment Pattern

A- Client-Side

In the RMI-HRT model the clients are assumed to be connecting to the remote server using sockets that are created by a socket factory class, where this socket factory must be configured during the initialization phase with the network address and the port number at the local machine, also as the client needs to connect to the server to make the call, then another pair of the network address and the port number of the remote server has to be assigned to the client. The configuration pattern that works as an extension configuration pattern of the remote communication model is required at the client side. In the HRT-RMI, this configuration pattern is provided by the `RMIHrtClientCfg` class, which defines a basic set of methods to set and retrieve these parameters as follows:

- **`ClientRMIHrtNetworkParameters getClientRMINetworkCfg(String refName)`**

This is a static method responsible for loading the network parameters of the remote object named `refName` from the class file that specify these configuration parameters. The function has first to check the if the file is loaded, if not; it loads it first using the `loadCfgCls()` method.

- **`void loadCfgClass()`**

This is a static method that loads the class file that defines the networking parameters of the client side in order to make the connection with the server.

- **`void loadRMIHRTClasses()`**

This is a static method that loads and initiates all the classes used by the middleware at the client side. The method is called in the initialization phase to minimize any external interference to the remote method execution by avoiding the dynamic class loading of the basic classes of the middleware in order to have better predictability.

- **`void loadSerClass(String stubName) throws RemoteException`**

This is a static method that is used to load the classes used for the predictable serialization model proposed in the HRT-RMI at the client side.

B- Server Side

As the remote object has to have a predictable behavior at the server in order to work in real-time distributed systems, then this remote object has to be assigned both scheduling parameters and network parameters during the initialization phase and before the mission phase. As the Java RMI, on which the RMI-HRT has been based, has no ability to support the configuration of the RMI to use such parameters the design of the HRT-RMI needed an extension pattern to support this feature either using a server centric approach, or a client propagated approach. In the server centric approach the parameters are saved at the server side and loaded at run time, where in the case of the propagated parameters approach, the parameters propagates from the calling client to the server side before executing the call.

In order to configure the server-side of the RMI-HRT middleware an extension pattern for configuration which uses the server centric approach has been used, where two types of parameters have been defined:

- Network Parameters; to specify the networking identification of the remote object within the network, This include the following:

1. The name of the remote object.
2. The server socket factory class used to create server socket classes.
3. The configuration parameters used by the server socket factory, including the network address and port number on which the server will be listening for the incoming calls.

- Real-time Parameters; these parameters control the behaviour of the remote object, these parameters include:

1. Sporadic parameters for the realtime thread responsible of service handling at the server, these parameters include:
   - The name of the remote object.
   - The listener priority
   - The maximum number of clients
   - The Memory Size of the scoped memory allocated for this real-time thread.
   - A list that holds the parameters of each client registered with this server.
2. Sporadic parameters of the real-time thread of the call handler that executes the call on behalf of the client, where each client has its own parameters registered with the server. For each client, these parameters include:
   - The reference name of the client.
   - The priority parameters to be assigned to the real-time thread executing the method on the server.
   - The minimum inter-arrival periods between each two successive calls to the same method.
   - The deadline of the execution of the remote method on the server.

The server-side configuration pattern of the HRT-RMI middleware is modeled in the `RMIHrtServerCfg` class that provides generally a set of methods that can be used to assign and retrieve the set of parameters that have been mentioned above.

The class `RMIHrtCfgClass` plays a major rule in loading these parameters during the initialization phase. Where this class defines the following methods:

- **`ServerRMIHrtNetworkParameters getServerRMIHrtNetworkParameters (String remObjName)`**

This is a static method responsible for loading the network parameters of the remote object named `remObjName` from the class file that specify these configuration parameters. The function has first to check if the file is loaded, if not, it loads it first using the `loadCfgCls`() method.

- **`ServerRMIHrtParameters getServerRMIHrtParameters (String  remObjName)`**

This is a static method that does a similar job to that of the `ServerRMIHrtNetworkParameters` method except that it loads the real-time parameters from the class file holding them instead of the network parameters.

- **`ClientRMIHrtParameters getClientRMIHrtParameters (ServerRMIHrtParameters remObjName, String client_ref)`**

This is a static method that loads the real-time parameters of the client that is registered with this object by the reference name `client_ref`.

- **`void loadCfgClass()`**

This is a static method that loads the class files that define both the networking parameters and the real-time parameters of the server side.

- **`void loadAllRMIHRTClasses()`**

This is a static method that loads and initializes all the classes used by the middleware at the server side. This method is called in the initialization phase to minimize any external interference to the remote method execution by avoiding the dynamic class loading of the basic classes of the middleware in order to have better predictability.

- **`void loadSerClass(HrtRemote obj) throws RemoteException`**

This is a static method that is used to load the classes used for the predictable serialization model proposed in the HRT-RMI at the server side.

7.3.3 Marshaller

As presented in earlier in this chapter, the Marshaller pattern is responsible for transferring the request parameters and results between the client and the server by converting them into raw bytes to be transformed over the network, and then reconstructing them back into their original form.

In the RMI-HRT, the marshaller pattern is a modification of the Java RMI marshaller pattern. So, in the following section, we present an overview of the marshalling pattern model in the Java language, and then we provide the modified pattern used by the RMI-HRT.

A- Marshalling in Java

In Java language the general marshalling pattern is provided in the Java object serialization process. Where the pbject serialization and deserialization was defined in the Java Object Serialization Specification (Sun Microsystems Inc.) as follows:

> *"Object Serialization in the Java Systems is the process of creating a serialized representation of objects or a graph of objects. Object Values and types are serialized with sufficient information to ensure that the equivalent typed object can be recreated. Deserialization is the symmetric process of recreating the object or graph of objects from the serialized representation"*

Java RMI wire protocol relies on the Java Object Serialization as a marshaller pattern in order to pass objects between Java virtual machines, where the object serialization produces a stream with information about the Java classes and objects that are being serialized over the network.

There are a set of requirements in order to make the serialization process. In the following we will identify the basic requirements:

1- The serializable class has to have a no-argument constructor to be used for recreation of the object in the deserialization process, where the a new Object is created with a null value of all its fields then the state of the field is restored by reading the values of the fields from the stream.
An object that implements the `OutputStream` is required for serializing the bytes to the stream, and an object that implements the `InputStream` is needed to deserialize the bytes from stream.

An object of the `ObjectOutputStream` class is required for serializing the bytes to the stream, where it provides the `writeObject()` to write the a Java object to the stream. Also, this class implements the methods defined by the `DataOutput` interface in order to write the primitive data types to the stream.

An object that implements the `ObjectInputStream` is needed to deserialize the bytes from the stream, this class provides the `readObject()` method to read a serialized Java object from the stream. Also, this class provides the methods of the `DataInput` interface in order to read the primitive data types from the stream.

In order to identify the classes that implement the serialization protocol, these classes have to declare either:

The `java.io.Serializable` Interface. This interface provides two methods: `writeObject(java.io.ObjectOutputStream out)` to serialize the fields of the object to the stream, and `readObject(java.io.ObjectInputStream in)` to deserialize the same fields back from the stream into a newly created object. Using this interface, the object serialization/deserialization is done automatically when reading or writing the serializable object using a default defined mechanism. This mechanism can be customized by overriding the implementation of the `readObject()` and `writeObject()` methods. Moreover, fields to be serialized can be optionally identified explicitly by adding them to the static member `serialPersistentFields` as objects of the type `ObjectStreamField` that define pairs of a name and a value for each of the added fields, or they can be discarded of the automatic process by declaring them as transient fields.

- The `java.io.Externalizable` interface**.** This interface defines two methods: `writeExternal(ObjectOutput out)` to save the state of the object and `readExternal(ObjectInput in)` to restore the state of a serialized object. By implementing this interface, the developer has complete control of the format and the contents of the serialized stream.

B- Serializing Class Descriptors

The class descriptor provides information about the serializable class including its fully qualified name, a unique serialization version of it, and its fields and their types. The class descriptor is represented in the class `ObjectStreamClass` which defines method to get this information.

In object serialization, when an object is serialized to a stream, information about all its super classes is saved with it, in order to re-instantiate it during the deserialization process. As defined in the serialization protocol in (Sun Microsystems Inc.), before writing the contents of each object into the stream, the class descriptor for the class of this object is written to the stream, where this class descriptor is written only the first time it is referenced, then a handle of it will be used. The writing to the stream is done by *recursively* calling the `writeObject()` in order to write the full object graph of the class where the sequence of writing of the information of the class will be as follows (Sun Microsystems Inc.):

1- The class name
2- The class modifiers
3- The names of interfaces (sorted alphabetically)
4- For each Serializable field (sorted alphabetically):
    a. The name
    b. The field modifiers
    c. The descriptor of the field
5- if class initializer exists
    a. The name `< cInit >`
    b. The method modifier `java.lang.reflect.Modifier.STATIC`
    c. The descriptor, `() V`
6- For each non-private Constructor (sorted by name and signature)
    a. The name
    b. The method modifiers
    c. The constructor descriptor, `() V`
7- For each non-private Method (sorted by name and signature)
    a. The name
    b. The method modifiers
    c. The method descriptor

C- Marshalling in RMI-HRT

The RMI-HRT model design aims to obtain a predictable memory management model as many real-time systems have limited resources. Hence, reducing the required memory is a primary goal for this middleware. However, as stated before, the marshalling in Java RMI is implemented by using object serialization and deserialization mechanism which is a heavy memory consuming activity as it uses complex recursive algorithms to handle references, and it uses reflection and polymorphism to discover information about classes of objects to be serialized. Hence, in addition to its being not compliant with the RTSJ memory model, this complexity makes it difficult to determine the worst case execution times and memory usage of the serialization process. As a consequence, and according to the requirements of the application making the remote call, the RMI-HRT model proposed two models of serialization (Tejera, Alonso et al. 2007):

1- **Real-time Serialization;** This model does not consider optimizing the serialization timing as it is proposed for the interaction with clients that have no critical constraints within the system. So, the RMI-HRT adapted the Classpath implementation in order to make it compliant with the memory model of the HRTJ profile, by enforcing the creation of all the temporary objects to be done in scoped memories, so that they are removed once the remote invocation is finished.

2- **Predictable Serialization;** This model aims to optimize the execution time and the memory usage of the serialization process in addition to making it compliant with the HRTJ profile.

The design of the predictable serialization model assumes that all the objects and classes being used in the realtime system are known in advance at the compilation time. Hence, in order to reduce the complexity, the serialization process is split in this model into two phases; compile time phase and runtime phase. Where the recursive processing made by the marshaller during the object serialization, to get the information about the classes of the serialized objects that executes dynamically during run time, is moved to be done statically by the *rmic* compiler during the compilation time; while, the writing of the object state is executed during the runtime. This split of the serialization process results in enhancing its execution time, and makes it avoids the allocation of many temporary objects during the run time, and makes it easier to determine its worst execution time. The implementation of the predictable serialization follows a class based strategy, where the classes used in each phase are as follows, see Figure 7-11:

- Compilation Time; at the compile time, two classes are created by the **rmic** compiler for each serializable class involved in the remote invocation in the system; a serialization class and a deserialization class, where these classes are used during runtime from within the stub and the skeleton, that are created as well at the compilation time by the rmic. The rmic compiler extracts the serialization information of the serializable class using the `RMICObjectOutputStream`, and it uses its method `writeClass()` to generate the serialization class. On the other hand, the rmic compiler uses the `readClass()` method of the `RMICObjectInputStream` to read the serialized information of the serialized class from the stream.

- Execution Time; at the execution time, the stub and the skeleton uses the `writeObject()` and `readObject()` methods respectively from the serialization classes that are generated during the compilation time, these methods internally access the byte stream using the methods of the `RMIHrtObjectOutputStream` and `RMIHrtObjectInputStream` where the methods of these two classes offer general methods to write to/extract the basic data types into/from the stream.

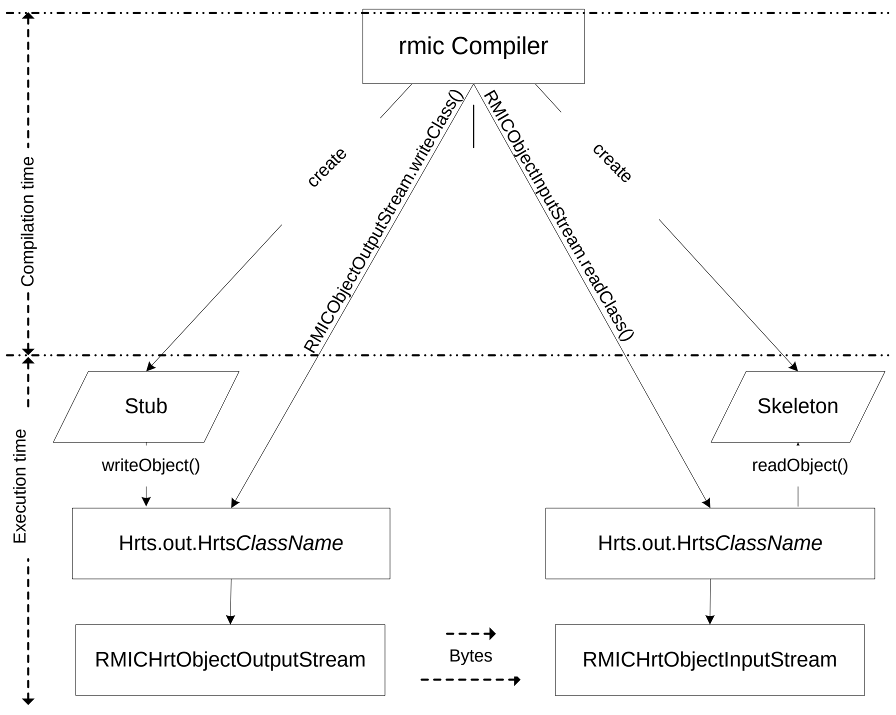


Figure 7-11 Predictable Serialization Classes

7.4 The Proposed Architecture of the RT-Middleware Model

The current architecture of the Java RMI inherits a lot of issues that inhibits using it in many real-time applications; we have provided an overview of these issues in chapter 3. In addition to these issues, the current RMI structure is based on the client server middleware paradigm, where the clients on a certain JVM can execute remote methods on the server objects residing on another JVM, the RMI specifications assumed the implementation of the model is using blocking networking I/O operations over the network stream. However, using the blocking networking I/O model, as explained earlier in this chapter, has a lot of deficiencies that affect the performance and predictability of invoking the remote methods that makes the Java RMI not suitable for use in many real-time applications.

To build our configurable real-time RMI server object, we designed our model over some of the classes of the RTSJ based RMI-HRT model presented earlier in this chapter. Our main change of the original model of the RMI-HRT is the addition of the support of the non-blocking I/O operations, through the use of the reconfigurable communicator component that we presented earlier in this thesis in chapter 5, where we assume that the client and the server objects are built as components, and the communicator component is a sub-component within the container of these components, which provide the low-level communication services, that are used by our RMI implementation within these, where the remote server object is encapsulated within the server components, whereas the client components encapsulate stubs of these server objects, see Figure 7-12.

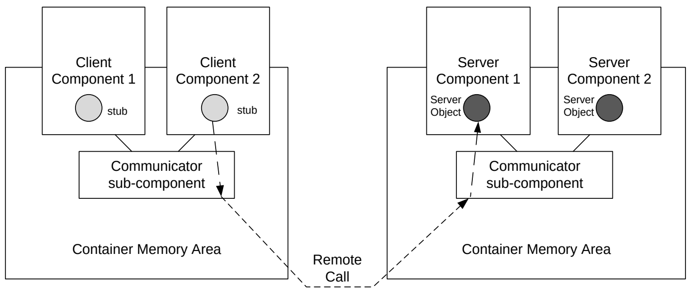


Figure 7-12 Proposed Model for the RMI-HRT Implementation

As we propose that the hierarchy of the RMI implementation will be within the component, and it uses the services of the communicator sub-component, then the implementation of this model will be sharing the forked memory model of the component and can use all the services provided in the container of these components as will be explained in this chapter.

7.4.1 Server Side

In our design of the server object of this proposed model, we assumed that the configurable communicator component that we presented in the previous chapter will be responsible for handling the incoming concurrent method calls coming to the server object/component. According to this assumption, the communicator component is required to implement the server side real-time request handler pattern of this middleware model. Hence, to achieve this requirement, our component has replaced the elements within the grey dashed frame shown in Figure 7-13, which represents the server request handler in the RMI-HRT model. The block diagram in Figure 7-14 shows our proposed model for the server object. In this new server object model, the component functionality provides the implementation of the request handler pattern in the RMI protocol.

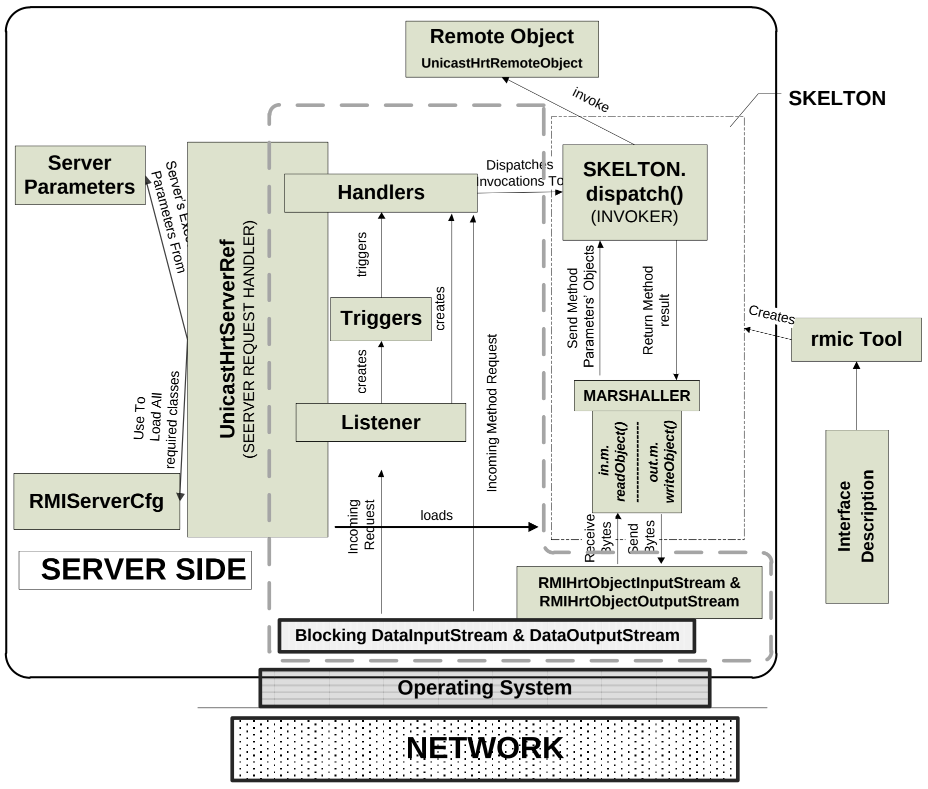


Figure 7-13 Changed Patterns of the HRT-RMI

The attachment of the communicator component to the server object is done in the ‘export’ process of the remote server object, which is done by `UnicastHrtServerObject.exportObject()` method, where a reference to the communicator sub-component object is passed to the `UnicastHrtServerObject.exportObject()` method; inside this method, the following actions are taken:

1- The component’s `createServerChannel()` is called so that a server socket channel is created within the server in order to be responsible of listening to the incoming requests on the specified port on the server. This approach replaces the old RMI model’s approach of creating a listening server socket.

The server socket channel is registered for the accept operation on the selector, this operation ensures that no individual thread is created and bounded to listen to the incoming requests, as the registration of the accept events of the server channels with the selector guarantees that the arrival of a request to the server channel will cause the operating system to notify the component’s selector element which in turn directs one of the component’s acceptor-executors to handle the call, as will be discussed later in this chapter.

The communicator sub-component can be used by several server objects/components, which coexist in the same container, concurrently as a common networking I/O interface because the component structure provides multiplexed I/O operations as discussed in the previous chapter. The diagram shown in Figure 7-15 shows our proposed model of sharing the communicator component with multiple server objects. The basic ideas of this model are that:

a. In order to be accessible by clients, the process of the exportation of each individual server object is modified, so that the exportation process creates a single server channel within the communicator component, where the network address and port number corresponding to this server will be published to the clients, e.g. in our model we assume that each individual server object/component is assigned port number and network address at design time. So, the server channel that is assigned to each server object/component will be responsible for receiving the connection requests coming from clients component, that want to access any of the remote methods in this server object.
b. The communicator sub-component assigns an executor element (one or more than one executor element as will be discussed in detail later) for each incoming method call from a client, where this executor(s) will be responsible for decoding the received call and receiving its passed parameters, then executes it at the server, and finally returns the result back to the calling client.
c. The component internally creates a new client channel for each single incoming method call coming from a certain client component, so that all the data exchange from the server object to/from this client is done through the client channel assigned for this specific client component.

According to this server model, shown in Figure 7-15, during the run time, each server object/component will be attached to the shared communicator sub-component through the skeleton of this object, so that the server object has a server thread within it, that is observing the events/requests coming from multiple clients concurrently, and a set of client channels equal to the number of the concurrent incoming/executing remote calls, in addition to a set of concurrent executors that are responsible for the processing of the concurrent calls.

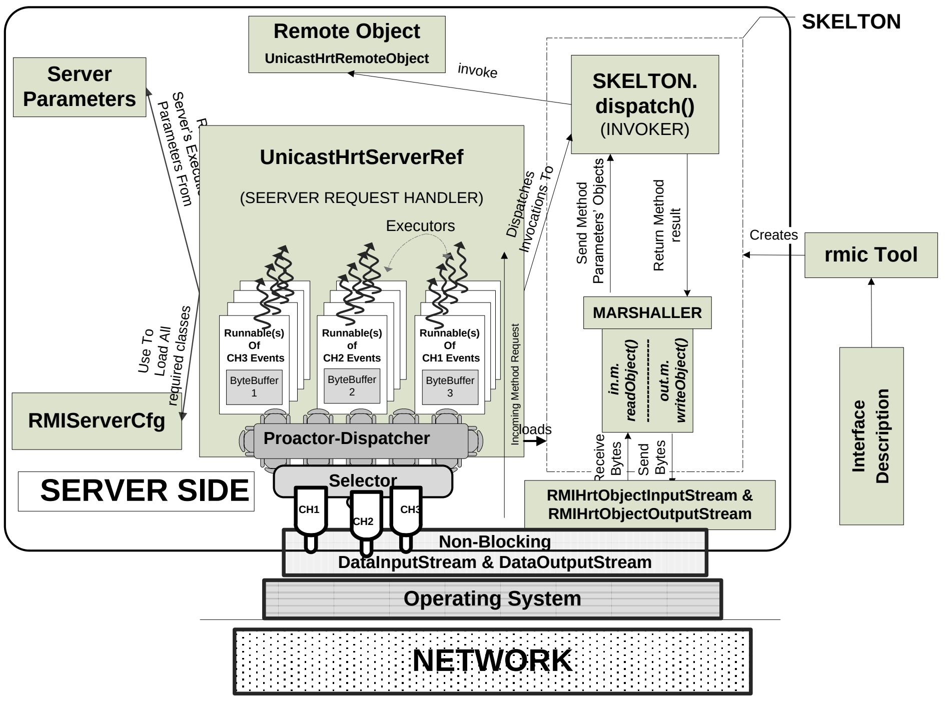


Figure 7-14 Patterns in the new Server Object Model

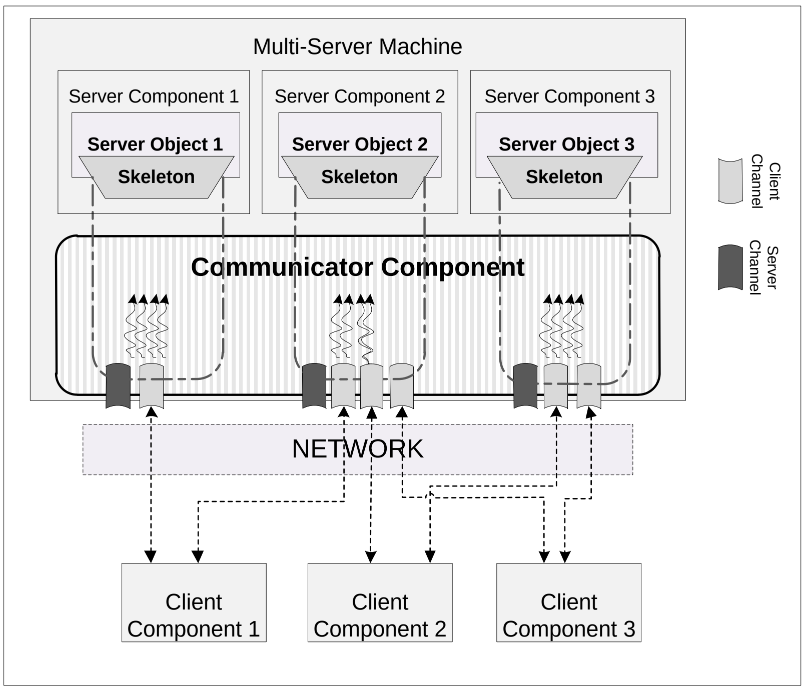


Figure 7-15 Sharing the Communicator Component among several Server Components

As discussed earlier in this thesis, the component's properties have to be configured to specify the middleware requirements that are available on both the machine and the operating system on which this middleware is running over (e.g. Selector Type property). However, some other properties will be dependent on the requirements of the middleware model itself in order to provide a predictable execution within the middleware model. So, the communicator sub-component has to be assigned minimum values for a set of its properties in order to provide predictable execution of its model. For the server side of our own middleware model, the communicator is assigned to be working in a *server* mode, where the following set of the communicator component's properties are important to be assigned minimum values defined by the model:

- **Number of Server Channels**; this property defines the maximum number of server channels that can be provided by the component, according to the above model, this number should have a value that is at least equal to the maximum number of server objects that are sharing this component.
- **Number of Client Channels;** this number is the maximum number of the client channels that are created to handle the concurrent method calls. Hence, the value of this property has to be set to the maximum expected concurrent remote calls received by all the server objects sharing this component.
- **Number of Executors in the Executors Pool;** the assigned maximum number of executors within the component is dependent on the model of handling the calls as will be explained in details later in this chapter, i.e. is it a single executor for handling each call or multiple executors for handling each call. Some possible assignments strategies for this property are:

1- The value assigned to this property is to be equal to the number of the server objects sharing this component, so that an executor can be servicing each one of the server objects sharing this component.
Another strategy is to assign it to be equal to the maximum number of concurrent remote calls serviced by the communicator in case of using a single executor for handling each incoming remote call.

As our communicator component can be configured to work in various ways as explained in this thesis; hence, different configuration of the communicator can provide different modes of operations of the server object. In later sections of this chapter, we will discuss these different modes and how they integrate with the client objects/components to implement the remote communication protocols.

7.4.2 Client Side

The communicator sub-component is built to support multiplexed communication over the network; hence, it can be shared by multiple client objects in the same time to make calls to various remote server objects. The diagram in

Figure 7-16 shows the basic model of sharing the communicator component among several client components. In this model, a client channel is created to connect with each remote object accessible by this client object/component, where each one of these channels becomes the I/O interface used by stubs created within each client component to connect remotely to one of the remote server objects accessible by this client. Hence, the Communicator provides for each individual (or shared) stub of all the client components the required executor(s) that handles the events occurring on this channel. The collaboration of each client channel and its associated executor(s), that are provided by the communicator sub-component, with its attached client component; through the stub of this client object, will be explained in detail in the incoming sections of this chapter.

According to the above model, the communicator component acts as a provider of the networking I/O communication interface at the client side for all the client objects running in the same container and as it was in the case of the server side, the component can be configured to process the I/O networking operations through its selectable channels over the network in multiple models (including the blocking and the non-blocking models) in order to provide a configurable real-time middleware that can be reconfigured to integrate with the target operating system to achieve the requirements of the applications running over it.

In addition to using the communicator component as the networking I/O interface at the client side of our model of the remote communication middleware, the model of the client side is required to provide the support of different invocation patterns as an additional functionality, that is not needed at the server side, but it is required at the client side to offer more flexibility of our proposed real-time middleware model. Hence, we assume that the client side in our real-time middleware model is responsible of enabling different method call invocation patterns other than the synchronous pattern. In this case we can consider the communicator sub-component to be working as a local server at the client side, where it receives the calls from the client objects/components, and send them to the target server on behalf of them, then when the result of each one of these calls returns to it, the communicator sub-component's executors will be responsible of delivering it to the client, either immediately by sending call-back calls to the client component that initiated the call or, indirectly by saving the result internally until the component client object requests it.

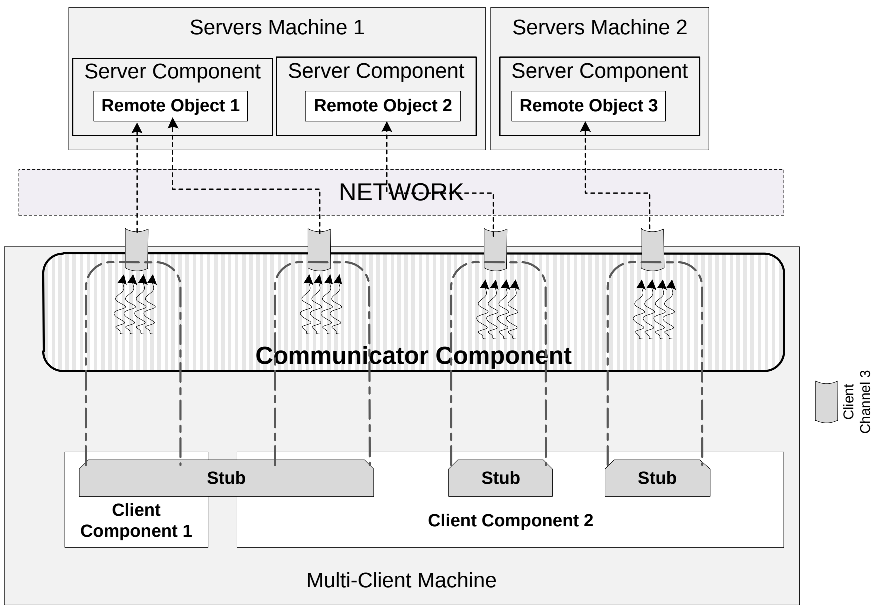


Figure 7-16
Sharing a single communicator sub-component by multiple clients

In order to support the above model, the communicator component is attached to each stub of the calling objects/components on the same container, to work as a client request handler for each of them at the client side. The diagram shown in Figure 7-17 shows the architecture of integrating the communicator sub-component with the stub within the calling object/component. As shown in this diagram, the interaction and collaboration between the communicator component and the stub occurs in several steps and through multiple elements, in order to transfer the call to the remote server object and then receive the result coming out of executing the remote method, to deliver it to the calling object/component, these steps and the elements constituting in their execution are summarized here as follows:

1- When a remote method is required to be called from a thread running in one of the client objects/components, the stub class of the remote server object that holds this remote method is loaded and an object of it is created where a reference to the communicator component object is passed to it in order to be able to access the component from within the stub.

2- If another component or thread in the same container wants to access the same remote method, or another remote method that exist with it in the same remote server object/ component, the same stub object is used to access it, so that the stub should be shared among all client objects/components that access remote methods on the same remote server object. This requirement means that the stub object should be created in a memory that can be accessed from within all the client objects/components.

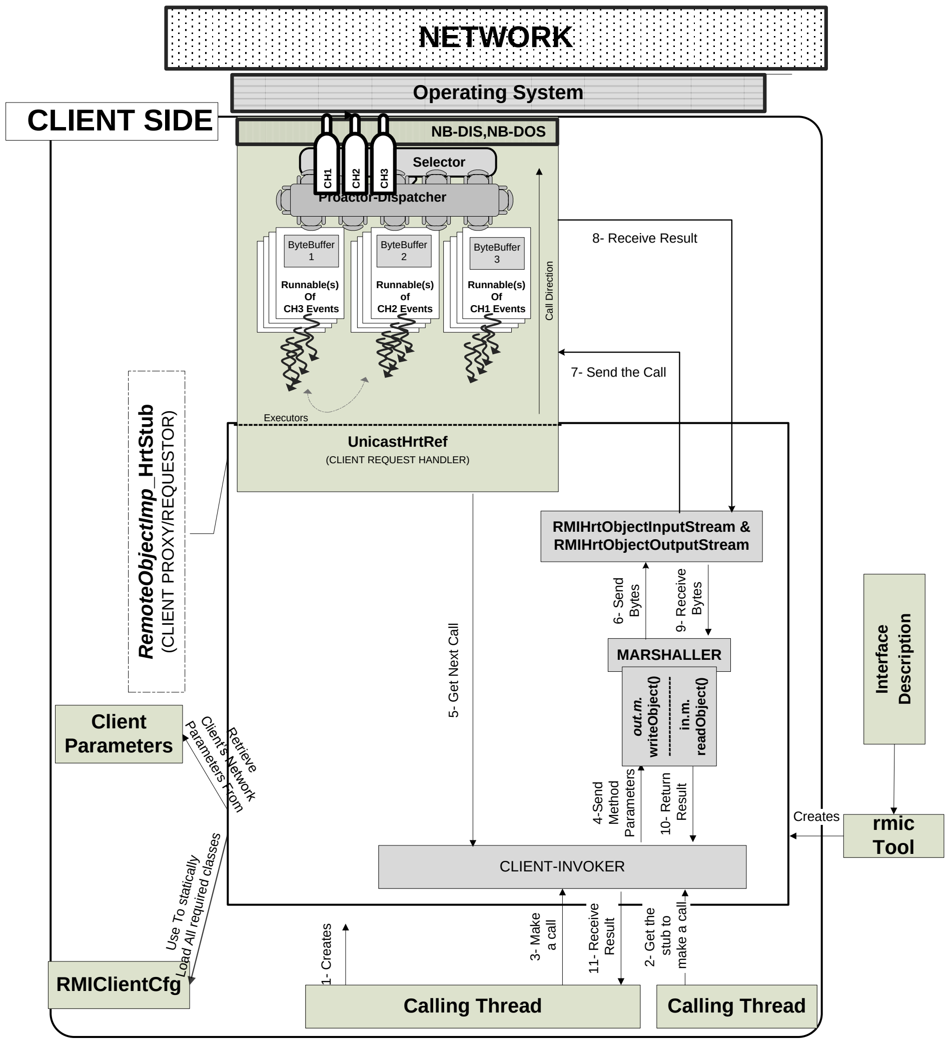


Figure 7-17 Communicator Component at the Client Side

3- To map the above requirements to the component memory model defined in chapter 5, we can define the following scenarios for creating the stub:

– **Creating the stub in the Heap;** this option is not suitable for real-time applications, especially when using the NHRT Threads.

– **Creating the stub object in the Immortal memory;** this option again is acceptable if the stub object is guaranteed to stay alive as long as the program is running, this is suitable for many distributed real-time systems as the remote methods is expected to be used throughout the life time of the application. But it is not efficient in case of short-life time components as the original stub object will not be deleted from the immortal memory of the same JVM and a mechanism for recycling the stub object may be required.

– **Creating the stub object in the container memory area (ContMA);** this is an optimum solution for sharing the stub among several components sharing the same container. This stub object would have a life time equal to that of the container itself. In this case the stub can be created in the ContMA and saved in its multi-name object portal to be accessible by all the components within this container.

– **Creating the stub object in the container memory area (CMA);** this can be used when a single component within the container is using this stub. This stub object would have a life time equal to that of the component that holds it. In this case the stub can be created in the CMA of the component and saved in its multi-name object portal to be accessible by all the threads running within this component.

– **Creating the stub object in the task's scoped memory stack;** where this scoped memory can be the SMA memory of the task using this stub or any nested scoped memory within it. This method is more suitable when the use of the remote methods is not frequent, and it is not needed to keep the stub alive for the duration of the component.

4- After getting a reference to the stub, the calling thread initiates the call to the remote method by calling the corresponding method in the stub which is responsible for executing it by delivering the call request to the Client-Invoker defined in the stub.

5- In our model, we adopt the Future call pattern. So, the Client-Invoker retrieves the remote call's data and its parameters and it calls the marshaller to process them. The Marshaller uses the predictable serialization mechanism provided in the RMI-HRT by using its serialization classes to convert the call's data and parameters in an encoded buffered form that is added to the queue that holds all the buffered calls that are ready to be processed later, where this buffered form can be saved in one of the memory area defined above so that it will be accessible by the stub.

6- The assigned executor for this client is retrieved from the communicator component, where usually this executor is the handler executor that is created within the communicator component during the connect operation.

7- The executor processes the calls queued in the queue, where it retrieves the buffered data of the next method to be processed from the queue in order to send it, where the call is already kept in the queue in a serialized form. The executor can be configured to execute the calls of the queue according to a policy defined by the developer, i.e. priority based, or FCFS, etc.

8- The bytes are sent to the remote server object through the methods of the `RMIHrtObjectOutputStream` class.

9- After receiving the parameters of the call and executing it at the server side, the return object/value is sent back to the client in a serialized form.

10- The component's executor assigned to the calling stub, receives the bytes through the methods of the `RMIHrtObjectInputStream`, and delivers them to the marshaller.

This executor can be the same one that used before for sending the call (in the case of blocking mode); or it can be another one, i.e. that is created as an executor handler for the read event that fires when the data arrives to the channel (in the case of non-blocking mode).

11- The marshaller deserializes the result into a real object/value and either:

- a- Saves it in an object known to the client, so that the client can retrieve the result either:
  - i. Immediately; in the synchronous mode, if the calling thread is blocked waiting for it, or,
  - ii. Later; in a future object pattern mode accessible by the client, when it, the client, requires it.
- b- The executor sends the result in a call-back method to the calling thread.

12- Finally, the calling client receives the result of the call.

As it was the case in the server side, the strategy of assigning and configuring some of the component's properties at the client side must be subject to the assumptions made in the designing the middleware model. Hence, at the client side, the communicator component is assigned to be working in a *client* mode, where we assume the following assignments for a set of its properties:

- The Number of Server Channels; this property is important even in the client-mode as it defines the port on which the selector would observe the events occuring on the registered client channels. But in the client-mode this value has to be set to value of one as we assume only a single selector is required to observe the events on all the client channels in the client-mode.

- The Number of Client Channels; the proposed model assumes the creation of a single client channel for each remote call made by a client object; hence, assigning a maximum value for this property is dependent on two main factors:

1. The number of client objects sharing this component.
2. The maximum number of concurrent calls made by each one of those client objects

According to these two factors, assigning the maximum number of clients will be dependent on both the invocation patterns used by the clients and the threading model of each client object, e.g. using synchronous invocation pattern with a single threaded model will enforce each client to use at most a single remote call at any moment; hence, the required number of client channels will be required to be at least equal to the number of the clients sharing the communicator component.

- **Size of Executors Pool;** like the previous property, this property is affected by both the mode of operation of the RMI model, and the invocation patterns used within it. For example, using a synchronous invocation pattern with objects running as single threaded, will require this value to be at least equal to the number of clients sharing the component.

In the next sections we will show the above operations in more detail, in order to show how this model integrates with the server object model to provide a predictable RMI model.

7.5 The Modified RMI Protocol

In order to build our real-time configurable middleware, we presented a modified version of the RMI protocol, this modified version uses our communicator component at both the server side and the client side, where the main modifications in this new protocol include:

1- Running the protocol in two phases, *initialization phase* and *execution phase*.
2- The use of selectable channels and selectors for networking communications instead of using the blocking sockets.
3- The use of the communicator component's pool of Executors as the executing elements of the call.
4- The addition of the ability to use either the server-centric approach or the client-propagated approach for assigning the execution parameters for executing the remote method at the server object.
5- The support of various invocation patterns other than the synchronous pattern offered by the Java RMI.

Our modified RMI protocol is divided into four main stages; *Handshaking, Message Handling, Parameters Retrieval and Assignment*, and finally *Call Execution*. In the following sections we will discuss the basic operations executed by both the client and the server in each stage in more details.

7.5.1 Handshaking

This is the first stage in the RMI protocol, where the client and the server exchange messages to check the compatibility of the calling client and the called server, in order to make the call. This stage starts by the client side that sends the following identifiers in sequence, see Figure 7-18:

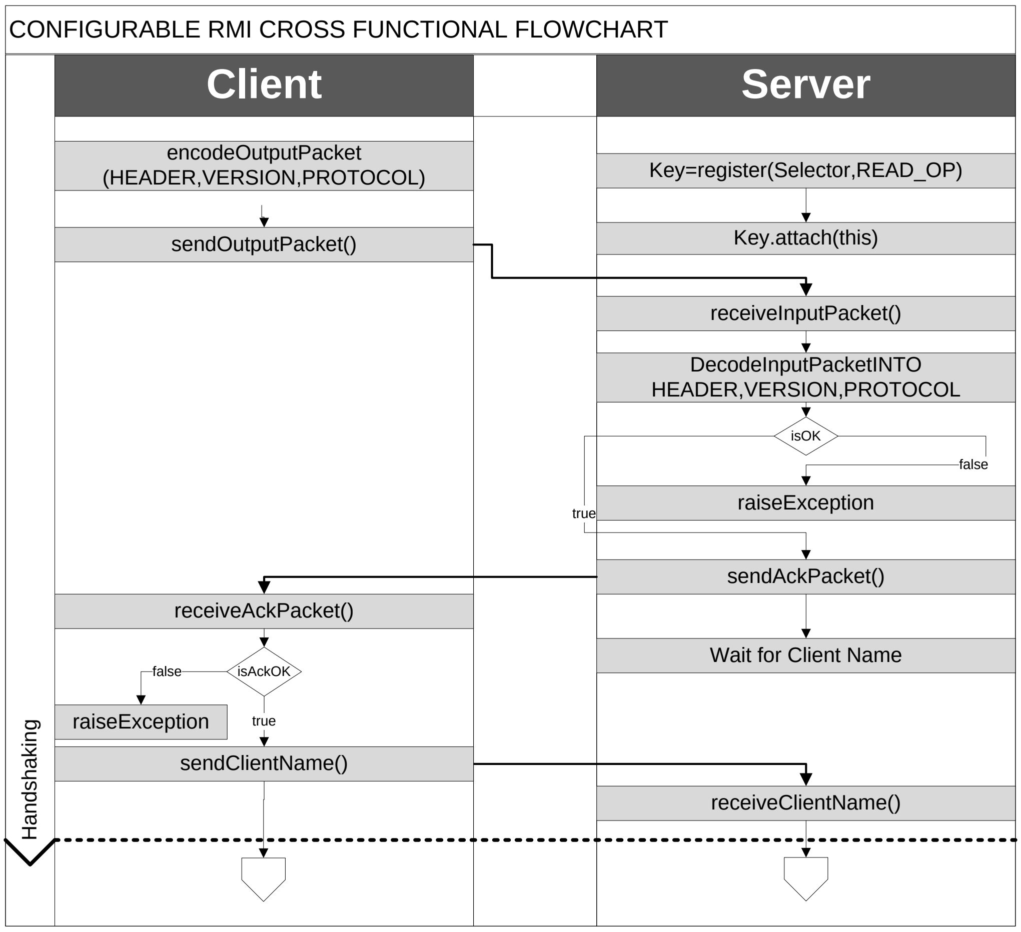


Figure 7-18 Modified RMI Protocol [Handshaking]

- The PROTOCOL_HEADER; this constant is a signature that identifies that this is an RMI call.
- The PROTOCOL_VERSION; this constant identifies the version of the used protocol.
- The PROTOCOL_TYPE; this is a type of the RMI protocol in use.

On the arrival of these messages at the server side, the handler thread at the server side reads them in the same sequence and checks the compatibility with the corresponding values defined at the server object. If they are compatible, the RMI protocol sends PROTOCOL_ACK acknowledge constant to the client and it continues to the next stage, otherwise it raises an `IOException`.

The main change that we made in this stage is the use of the communicator component's `NBDataWriter` and `NBDataReader` to write and read the exchanged constant message at both the client and the server instead of the blocking `DataInputStream.read()`, and `DataOutputStream.write()`. In our protocol, we assumed that the request handler at the client side writes the three message constants into its own byte buffer, then it passes it to the `NBDataWriter.write()` method to write them as one packet to the server. Hence, on the server side, the server handler will receive the bytes into its associated byte buffer as a single packet using the `NBDtaReader`.read() method, then it will encode the messages into the three original constants, in order to check the compatibility by comparing them with the values defined at the server.

7.5.2 Message Handling

The message handling stage is the third stage of our modified RMI protocol. In this stage an identifying message is sent from the client to the server, to identify the requested remote method. At the server side, see Figure 7-19, this stage starts with an intermediate waiting state, in which the server's call executor waits for the message to arrive from the client, that holds information about the required request, in order to handle it in the next stage. In order to do this, the call executor registers its interest of the read events that occurs on the channel with the selector, and attaches itself as a token to the selection key, so that it is notified when the message arrives to the channel. Once the message arrives from the client to the channel and it is ready to be read, the call executor is notified to start reading it. Two types of messages are expected to arrive from the client; otherwise an exception is thrown at the server side, the expected two message types are:

- PING Message: This message can be sent by the client to check if the server is still alive or not. The server replies to this message by sending a MSGPING_ACK message back to the server, and the call executor goes back to its initial state, waiting for the next client-request.
- MSGCALL Message: That indicates that the client is interested to make a remote call to one of the methods exported at this server object. In this case the server expects to receive another message packet that holds information of the required method.

In the case of the MSGCALL message type, once the message packet arrives from the client to the server, the server decodes the packet into the following:

- Method ID:
- Method Hash
- Protocol Magic
- Version

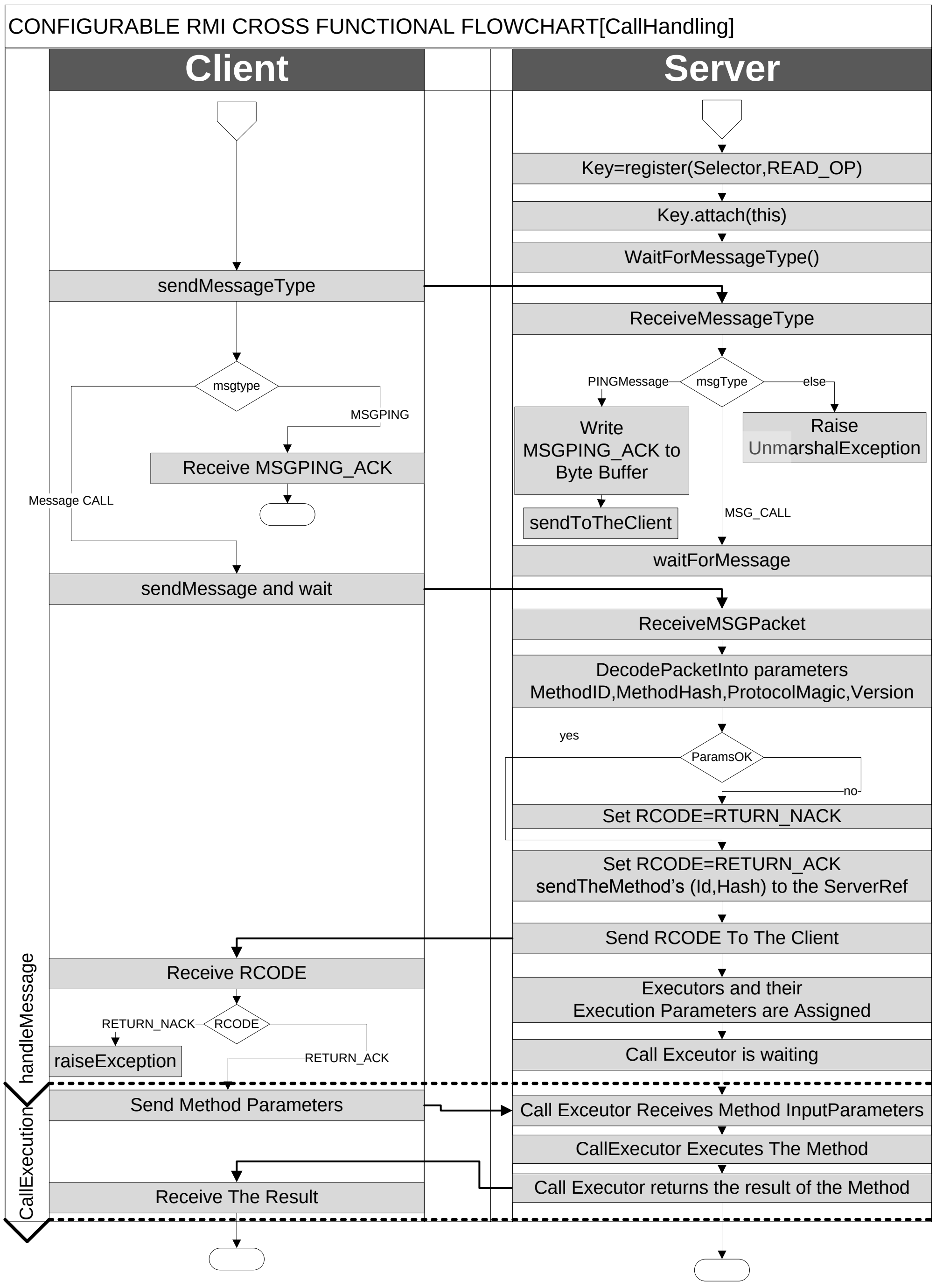


Figure 7-19 Modified RMI Protocol [Message Handling]

These parameters are checked by the server's call executor for compatibility and validity. If the check fails, the call executor sends back to the client a RETURN-NACK message to inform it of the invalidity of making the call; hence an exception is raised at the client side. Otherwise, if the decoded parameters are valid, the server sends a RETURN_ACK message to inform the client that it is able to execute the call, so that the client can start to send the method parameters; if there are any, in the following stage.

7.5.3 Parameters Retrieval and Assignment

In this stage, the execution parameters including the scheduling, release, processing group, and memory parameters are loaded and assigned to the executor that will process the execution of the remote method on the server object. There are three possible scenarios of the execution in this stage as follows:

1. **Fixed Server Centric Parameters;** do not load anything and use statically fixed set of default execution parameters defined for the elements of the server side.
2. **Loadable Server Centric Parameters;** load the execution parameters from a configuration file on the server to be assigned later to the processing executor.
3. **Client Propagated Parameters;** load the execution parameters from the client by extracting their configuration values from a packet sent by the server, and recreate objects representing them at the server side.

In our model, we assume that, for scenario 2 and scenario 3 above, transferring the construction values of the objects instead of serializing the objects themselves is important, particularly for the client-propagated approach, as using serialization to transfer them will increase the size of the packet with unnecessary data, as all the construction parameters have numeric values, which are relatively much less in size than the size of the objects of the execution parameters and their classes, that have to be transmitted if serialization is to be used, especially that the classes of these objects already exist at the server side as long as it supports the RTSJ, which is a principal assumption in our model. Moreover, none of the classes of the execution parameters (e.g. ReleaseParameters, etc.) is serializable which means either custom serialization is to be used to serialize them or their definition has to be changed to make them serializable. On the other hand, using the construction parameters of the objects requires theses parameters to be fixed and arranged in a certain predefined order known to the server, so that it will be able to extract their values when they arrived to it. In our model we assume the following variable length structure of the packet/file-record that holds the construction parameters, see an example in Figure 7-21.

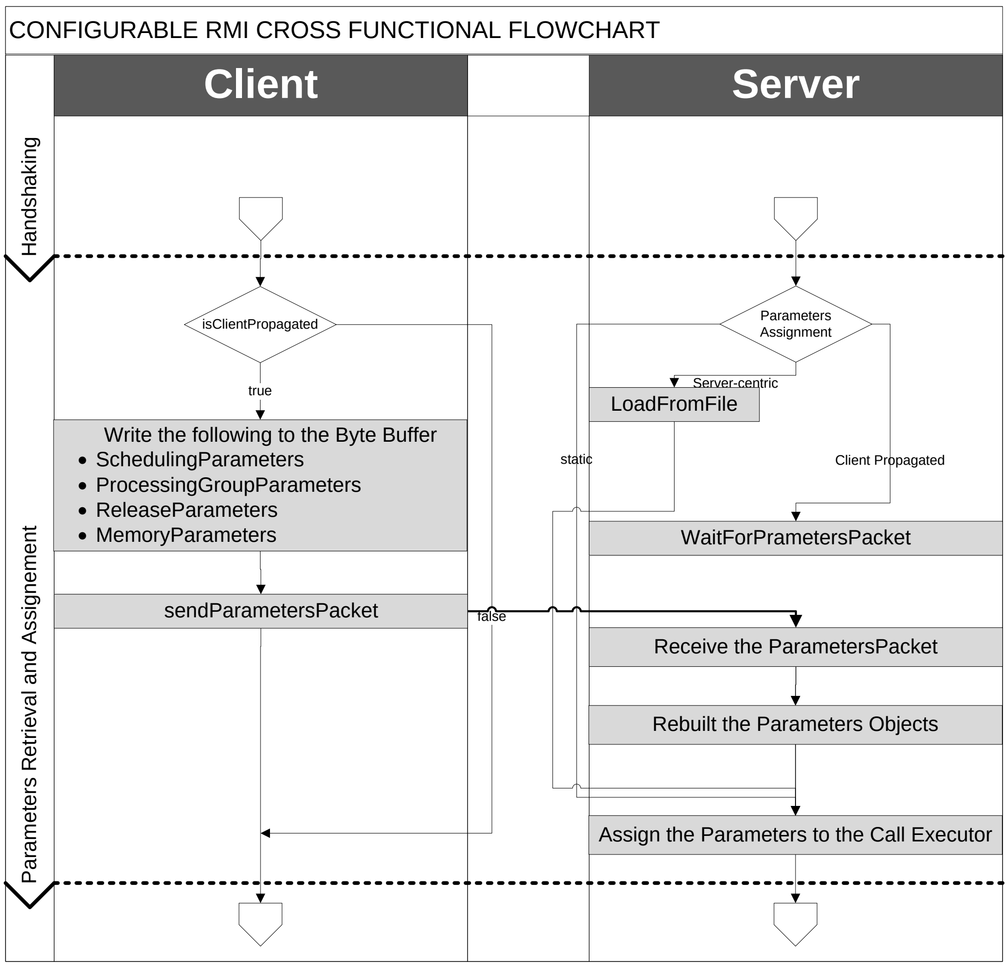

CONFIGURABLE RMI CROSS FUNCTIONAL FLOWCHART
Client
Server
Handshaking
Parameters Retrieval and Assignement
isClientPropagated
true
Write the following to the Byte Buffer
• SchedulingParameters
• ProcessingGroupParameters
• ReleaseParameters
• MemoryParameters
sendParametersPacket
false
Parameters Assignment
Server-centric
LoadFromFile
static
Client Propagated
WaitForPrametersPacket
Receive the ParametersPacket
Rebuilt the Parameters Objects
Assign the Parameters to the Call Executor

Figure 7-20 Modified RMI Protocol [Parameters Retrieval and Assignment]

1- The packet starts with a single **byte** variable, ***Sequence***, that is encoded to represent a predefined constants that specify the parameters sequence in the packet, e.g. the constant S-R-M specifies that the packet has scheduling, release, and memory parameters in sequence and there is no processing group parameters in it.

2- The second element in the packet holds is a **short** value that specifies the *remaining* length of the packet.

3- After that, each parameter included in the packet is represented in the packet by two basic elements:

a- The **Type** of the parameter. It is one of a set of predefined constants that refer to the type of this parameter.

b- The **Values** of the constructing values of the parameter. These are a sequence of the ***numeric*** values that can be used to construct the parameter object, where each value can have sub-values and each sub-value takes a size of the packet equal to the size defined for it in the parameter class.

For example, in the packet shown in Figure 7-21, the scheduling parameters are represented by its type constant PRI which refers to Priority Parameter Type, then the *Priority* value of the single numeric construction constant of this parameter follows its type. Whereas, in the same packet, the release parameters type is APER, which refers to aperiodic parameters and it is followed by its two numeric constructing values cost, and deadline where each one of the values is represented by two sub-values, which represent the *milli-seconds* and *nano-seconds* parts of respectively.

In order to build the above packet structure during the run-time at the client side, the call executor has to know what the parameters that will be sent are, and what their values are. Several scenarios can be used at the client side to get these parameters objects, examples of these scenarios are:

- The parameters can be retrieved from the execution parameters of the calling schedulable object itself by using the `getParametersTypeParameters()` family of methods, e.g. retrieving the priority of the calling schedulable object to be the priority of the call executor at the server thread, assuming similar priority levels at both sides of the remote call.
- The parameters can be loaded from external static file at the client side, which define parameters' values for each individual remote method.
- The parameters are passed with the method parameters to the stub's defined method when the remote method is called, e.g. as the first 4 parameters of any call, in this case the stub generation has to be able to retrieve the parameters assigned by the caller and isolate them from the remote method parameters, then it will be responsible for build the packet of this stage, then later it passes only the method parameters to the remote method.

In any of the above scenarios, the following algorithm can be used to build the parameters packet at the client-side:

- The RMI configuration is checked to see the configured sequence of parameters, and its code is written as the first byte of a byte buffer
- The parameters defined in the sequence are retrieved in order by one of the above approaches.
- The type of each parameter is detected, and its required length is calculated.
- The total length of the written values plus their types constants are calculated and saved in the packet as the remaining length of the packet; LENGTH.

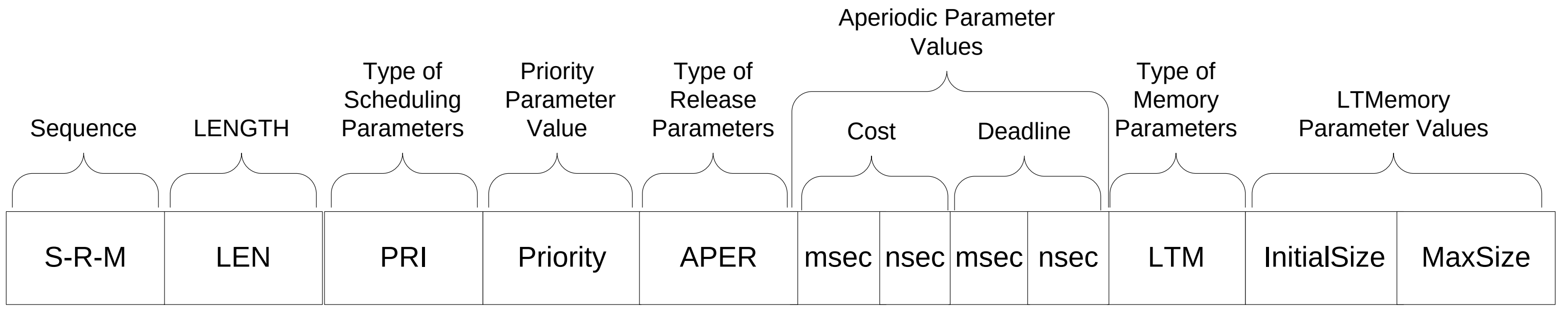

Aperiodic Parameter Values
Sequence
LENGTH
Type of Scheduling Parameters
Priority Parameter Value
Type of Release Parameters
Cost
Deadline
Type of Memory Parameters
LTMemory Parameter Values
S-R-M
LEN
PRI
Priority
APER
msec
nsec
msec
nsec
LTM
InitialSize
MaxSize

Figure 7-21 Example of the Parameters Packet

Each type constant is followed by values (or sub-values) that are written to the byte buffer.

- After building the packet, it is sent using the call executor through the channel over the network to the server. At the server side, a reverse process is made to rebuild the parameters and assigning them to the required call executor, which will execute the method call. The following algorithm represents this process:

1. The first 3 bytes of the packet is read, to check the sequence and the length.
2. The sequence is decoded to discover the received parameters types and their order.
3. The next LENGTH bytes are read, and then these bytes are decoded as follows.

    a- A byte that represents the type of the parameter is decoded according to its sequence order, hence, the length of its corresponding construction values (and their sub-values), is retrieved.

    b- The construction values (and their sub-values) are read.

    c- An object is created of this parameter type and initialized with these construction parameters.

    d- The steps a-c above are repeated for all the next parameters in the sequence.

4. After finishing the reconstruction process, the created objects, are assigned to the call executor that executes the call on the server.

7.5.4 Call Execution

The call execution stage is the most important stage in the protocol, in this stage, as seen in Figure 7-22, the client stub sends the set of the parameters that it receives from the client object/component to the remote machine in order to pass it to the remote method of the server object which is required for execution. According to our proposed model, an executor of the communicator component is used at the client side to send these parameters. On the other side, the server side, a call executor is responsible for receiving these parameters and passing them to the required method on the server object/component; then, it runs this method and waits for its returned result to return it back to the calling client object/component when it is ready. Once, the result arrives to the client machine, the client object/component, which called the remote method, in turn receives the result through an executor of the communicator component's to which it is attached.

From the above, we can see that in this stage the major elements contributing in it at both the server side and the client side are the communicator component executors. However, in our model of this component, we assumed different configuration of the executors running within it that can lead to different patterns of executions. Hence, in the following sections we need to study the effect of the different configuration of the executors on both the server side and the client side of the proposed real-time middleware model. Moreover, we will see how we can develop different invocation patterns at the client side, and how these patterns can be integrated in different scenarios with the RTSJ memory model.

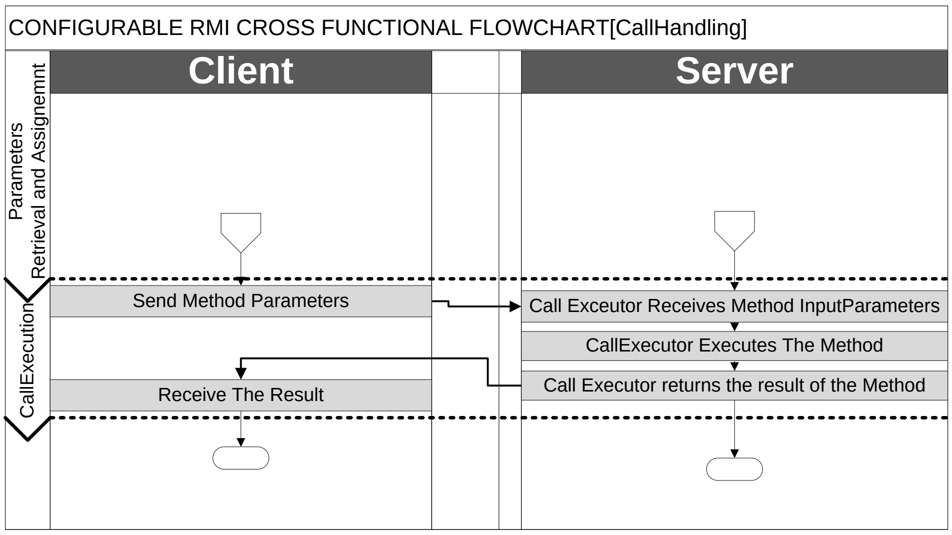

CONFIGURABLE RMI CROSS FUNCTIONAL FLOWCHART[CallHandling]
Client
Server
Parameters Retrieval and Assignemnt
CallExecution
Send Method Parameters
Call Exceutor Receives Method InputParameters
CallExecutor Executes The Method
Call Executor returns the result of the Method
Receive The Result

Figure 7-22 Modified RMI Protocol [Call Execution]

7.6 Models and Patterns of Executions in the RT Middleware

As mentioned in the previous section, the call execution at the client side and the server side is done using the executors provided by the proposed communicator components, and as this component can be configured to provide multiple execution patterns during its execution phase; hence, different models of execution can be used at both the server and the client side of our proposed real-time remote communication middleware. In addition, as both the memory model and the scheduling model of the RTSJ and the complexity managing them has led to a set of patterns that can be used to overcome this complexity then, using and integrating these patterns within the implementation of the real-time remote communication model provides a configurable architectural model that can be built and run using these patterns and models.

7.6.1 Models of Call Execution at the Client Side

The call execution at the client side cannot be discussed without considering the different possible invocation patterns that can be used; this is because in our model the actual invocation of the call is done by the executor(s) provided by the component and they are affected by how the component is configured to run these executor(s). Hence, according to the configuration of the number of executors running within the component to serve the remote calls made client objects, we can classify the following different models.

A- A Single Executor/Stub Model

In this model the single *connect* executor which is created in the communicator component to process the handshaking protocol with the server object/component continues its execution to handle all the calls that come from a single stub during the execution stage. This executor will be responsible for the execution of all the remote methods initiated from all clients to a certain remote server object/component which this stub acts as a proxy to access its methods. This model limits the concurrent access to the server object/component as only a single executor should process all the remote calls targeting this server object/component; hence, this model can cause unpredictable delay at the client side due to the blocking of the calls while they are waiting for the executor to be free to process them. In this model the execution of the calls is done sequentially, i.e. all the remote calls that have to be done through a single stub are saved in a call-queue associated with this stub and the executor processes these calls sequentially where it processes them according to a certain criteria, e.g. on the priority of the caller, or its arrival time, etc. In order to process the remote calls they can be represented as a byte array of objects that is processed by the executor

Due to it, the unpredictability that results for processing the remote calls sequentially, we assume that this model best suits systems that do not need concurrent access to remote objects.

B- Multiple Executors Model/Stub

In this model, in order to support parallel and concurrent processing of the calls, the communicator component provides more than one executor to process the calls along the lifetime of the call; two sub-models of this model can be built:

- Single Executor/Call; in this mode, the processing of each individual remote call through a single stub, is executed by a separate executor from the moment the call is ready to start untill its result is returned. This can be built into our communicator component by enforcing each instance of the *write executor,* that is activated within the communicator component to handle the readiness of the channel to start writing a call through it, to process all the operations sequence required to execute the call remotely. Although this model assumes parallel execution of multiple concurrent calls, but, from the single call point of view, it is still working in a blocking mode as the write executors used in this model cannot do useful work, while waiting for some data from the server object. Hence, it still has many of the cons of the previous model that can limit its use.

- **Multiple Executor/Call.** In this model, the created instances of the read executors and write executors, that are created over the channel on which the call is sent to the server as a result of readiness of read and write events on this channel, these executors collaborate together to process the requested remote call over the network. In this model, the logic of the call processing is defined as a state machine where the state machine of a single call is attached to the channel that is processing it, so that the executors can do the following:

1. Retrieve the state machine associated with the channel.
2. Validate the next state in the state machine and progress the state machine to be ready for next state.
3. The executor returns back to the pool of executors.

As seen from the above sequence, the executors do not block during the remote call processing, and they can do some other useful work. So, this model looks to be working as event-based system in a non-blocking mode. This model is considered to be much more suitable for real-time systems as its design does not include any blocking that can make the system unpredictable.

7.6.2 Call Execution at Server Side

According to the design of our communicator component model, the RMI operations have to be executed from within the runnable logic(s) assigned to the executors. However, in the RMI-HRT protocol the executor should forward these calls to the `UnicastServerRef` object, which in turn forwards them to the skeleton that executes the calls locally. To provide this RMI functionality to the executors, an extended class of the required executor type has to be created to include the reference to the server’s `UnicastHrtServerRef` object, e.g. the `rt_BAEH_RMIHandler` class in Figure 7-23 represents an extended class that is assigned to the communicator component’s executor type property.

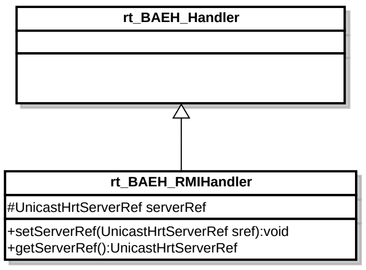

rt_BAEH_Handler
rt_BAEH_RMIHandler
#UnicastHrtServerRef serverRef
+setServerRef(UnicastHrtServerRef sref):void
+getServerRef():UnicastHrtServerRef

Figure 7-23 RMI Executor Class

According to our model, at the beginning of this stage, the call executor at the server side will be still running using the same execution parameters that it has been used since the request arrived to the server. So, we assume that according to the configurations of the component, as it was the case in the client side, the following scenarios can happen during this stage:

1. The same call executor will continue to handle the call.
2. A new call executor(s) will handle the call.

These scenarios can lead to different modes of operations of our real-time remote communication model. In the following sections we are discussing these modes.

A- Synchronous RMI Server

To build RMI server object running this mode, our communicator component uses a scenario in which one executor, the acceptor executor, will be responsible for executing the logic of all the RMI operations from the time the call is accepted, untill it returns the result to the client, i.e. the communicator component will be configured to be running in blocking mode, while the read and write events do not initiate separate executors, but when they occur, the acceptor executor is notified to continue executing the logic assigned to it, where the logic running by this acceptor executor is assumed to be divided into sections corresponding to the different stages of the call handling, where each section is preceded by blocking waiting statements to control the execution of these sections. The pseudo code of the acceptor executor's logic in the synchronous RMI server model is shown in Figure 7-24, where all the stages (e.g. Handshaking stage), and their sub-stages that are handling the call are encapsulated in sequence in a single logic, where the transfer from a stage, or sub-stage, to a next one is controlled by the arrival of bytes to the channel which results in a read readiness event that notifies the executor, which in turn moves to execute the next stage.

```
Start
+Do-Handshaking
        - Read MSG1
        - DecodeAndCheck MSG1
        - Prepare Reply of MSG1
        - Send Reply of MSG2
        - Attach this executor to the
        channel
        - Register Read Event on the channel
        - WaitForReadEvent
        - Receive MSG2
        - DecodeAndCheck MSG2
        - Prepare Reply of MSG2
        - Send Reply of MSG2
        - Attach this executor to the
        channel
        - Register Read Event on the channel
        - WaitForReadEvent
        - Receive MSG3
        ……….
        - Attach this executor to the
        channel
        - Register Read Event on the channel
        - WaitForReadEvent
+Send_Execution_Parameters
        - Send The Parameters Packet
        - Attach this executor to the
        channel
        - Register Read Event on the channel
        - WaitForReadEvent
+Send-Method-Parameters
        - Send the Serialized Parameters
        - Attach this executor to the
        channel
        - Register Read Event on the channel
        - WaitForReadEvent
+Receive-The-Result
End
```

Figure 7-24 Pseudo code of the logic of Synchronous RMI Server

The communicator component can be configured to execute the logic defined for the handshaking stage as defined in the above pseudo code easily by making it to work in synchronous mode as shown in Figure 7-25, where in the synchronous mode a reference to the `RMIExecutor` is passed at the end of both the non-blocking read and write operations to the `NBDataReader/NBDataWriter` object as an attachment, while the executor blocks waiting for the next registered *read* event, so that the next time, the server component receives a new read event from the same channel, it notifies the same executor to continue processing the RMI handshaking logic.

In the case of handling the call itself, the same technique is used; however, in this case the RMI Protocol's logic of the `RMIExecutor` directs the execution to the skeleton through the `UnicastHertServerRef` object. The deserialization and serialization operations run by the skeleton are themselves running as a sequence of:

```
ReadCallParameters=>WaitForCompletion=>
DecodetheCall=>ExecutetheCall()=>WriteReturnPacket
```

This sequence is similar to the handshaking sequences, with the difference that the packet size for the handshaking sequence is very small, but it is repeated more than once while, in the case of the serialization/deserialization sequence, the packet size is relatively much larger as it is dependent on the size of both the objects and the classes of the received remote method parameters and the returned result.

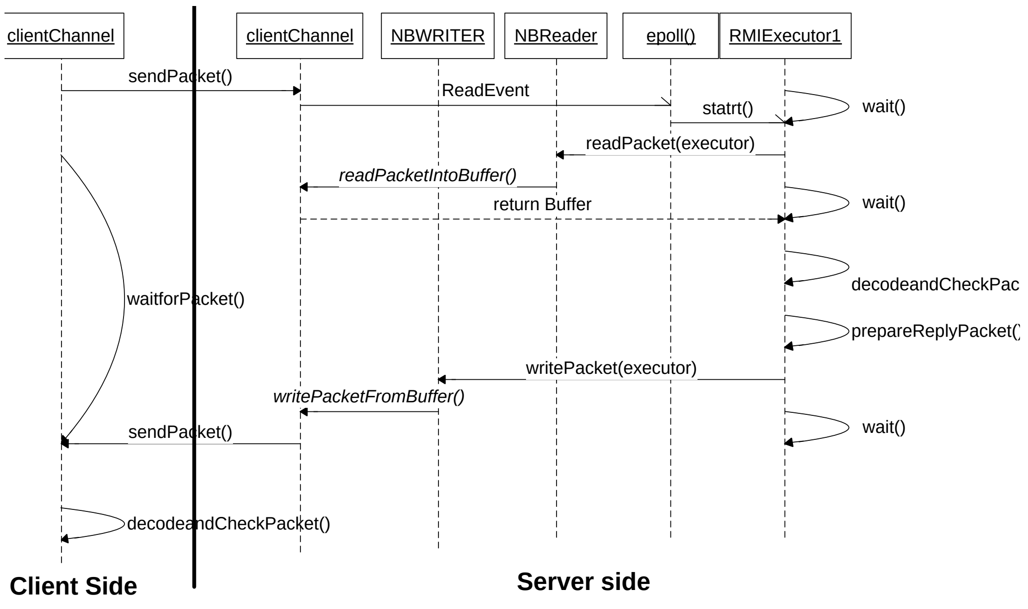


Figure 7-25 One Executor for all Requests Sequences

B- Non-Blocking RMI Server Model

The main difference in this mode, as shown in Figure 7-26, is the assumption that each request sequence of the RMI's protocol operations is executed by a different executor, this is to avoid keeping the same executor blocking between two consecutive request sequences.

According to this assumption, the logic of the server should be split among these executors, hence, the runnable logic of this server is best implemented as a state machine as shown in Figure 7-26, where the operations of the server are represented as states [STEP1, STEP2, ….] and the execution of these states are dependent on a control variable `nextState` which determines the state to be executed. This state machine logic is executed by all the executors that are used for handling the call at the server side;

hence, each time the logic defined by this state machine is executed by one of the executors it executes one state and before exiting, it registered the expected next event for the next state and also it sets the value of the `nextState` variable to the next state and attaches the runnable object that holds the code of the state machine to the channel used by this call, so that when the registered event occurs on this channel, its corresponding executor handler is started to handle the event by entering the attached runnable logic that defines the state machine, where it will execute the next state as defined by the `nextState` variable.

```
Start
If(nextStep=STEP1)
  +Do-Handshaking
        - if (nextStep=STEP1-1)
        - Read MSG1
        - DecodeAndCheck MSG1
        - Prepare Reply of MSG1
        - Send Reply of MSG2
        - Set nextStep=STEP2-2
        - Attach the logic of this executor  to the
        channel
        - Register Read Event on the channel
        - exit
        - if (nextStep=STEP1-2)
        - Read MSG2
        - DecodeAndCheck MSG2
        - Prepare Reply of MSG2
        - Send Reply of MSG2
        - Set nextStep=STEP1-3
        - Attach the logic of this executor  to the
        channel
        - Register Read Event on the channel
        - exit
- if (nextStep=STEP1-3)
        - Read MSG3
        ………..
        - Set nextStep=STEP2
        - Attach the logic of this executor  to the
        channel
        - Register Write Event on the channel
        - exit
If(nextStep=STEP2)
  +Send_Execution_Parameters
        - Send The Parameters Packet
        - Set nextStep=STEP3
        - Attach the logic of this executor  to the
        channel
        - Register Write Event on the channel
        - exit
If(nextStep=STEP3)
  +Send-Method-Parameters
        - Send the Serialized Parameters
        - Set nextStep=STEP4
        - Attach the logic of this executor  to the
        channel
        - Register Write Event on the channel
        - exit
If(nextStep=STEP4)
  +Receive-The-Result
End
```

Figure 7-26 Pseudo Code of the Runnable logic (State Machine) of the Non-Blocking RMI Server mode

To run the RMI server object in this mode, see Figure 7-27, the communicator component is configured to run in proactive mode. In this mode, as in the synchronous mode, a reference to the `RMIExecutor` is passed to the `NBDataReader` object as an attachment and the executor blocks waiting during the reading part of the operation. However, in the writing part of the operation sequence, a reference to the Runnable logic that is assigned to the `RMIExecutor`, is passed to the `NBDataWriter` object as an attachment in order to be registered with the channel. In the same time the current executor finishes its execution and returns back to the executors' pools. Hence, when the read event of the next request sequence occurs at this channel, the communicator checks the attachment, and as it is a Runnable object, it retrieves a new executor from the executors' pools and assigns this runnable logic to it, and then it starts its execution. In order to make the new executor continue the execution of the runnable logic after the last point executed instead of restarting the execution. The logic of the assigned runnable object must be defined as a state machine, where each request sequence is preceded by a check of a request-flag that determines if this request has been executed or not. All the requests-flags are initially unset, and after finishing the execution of each request its associated request-flag is set.

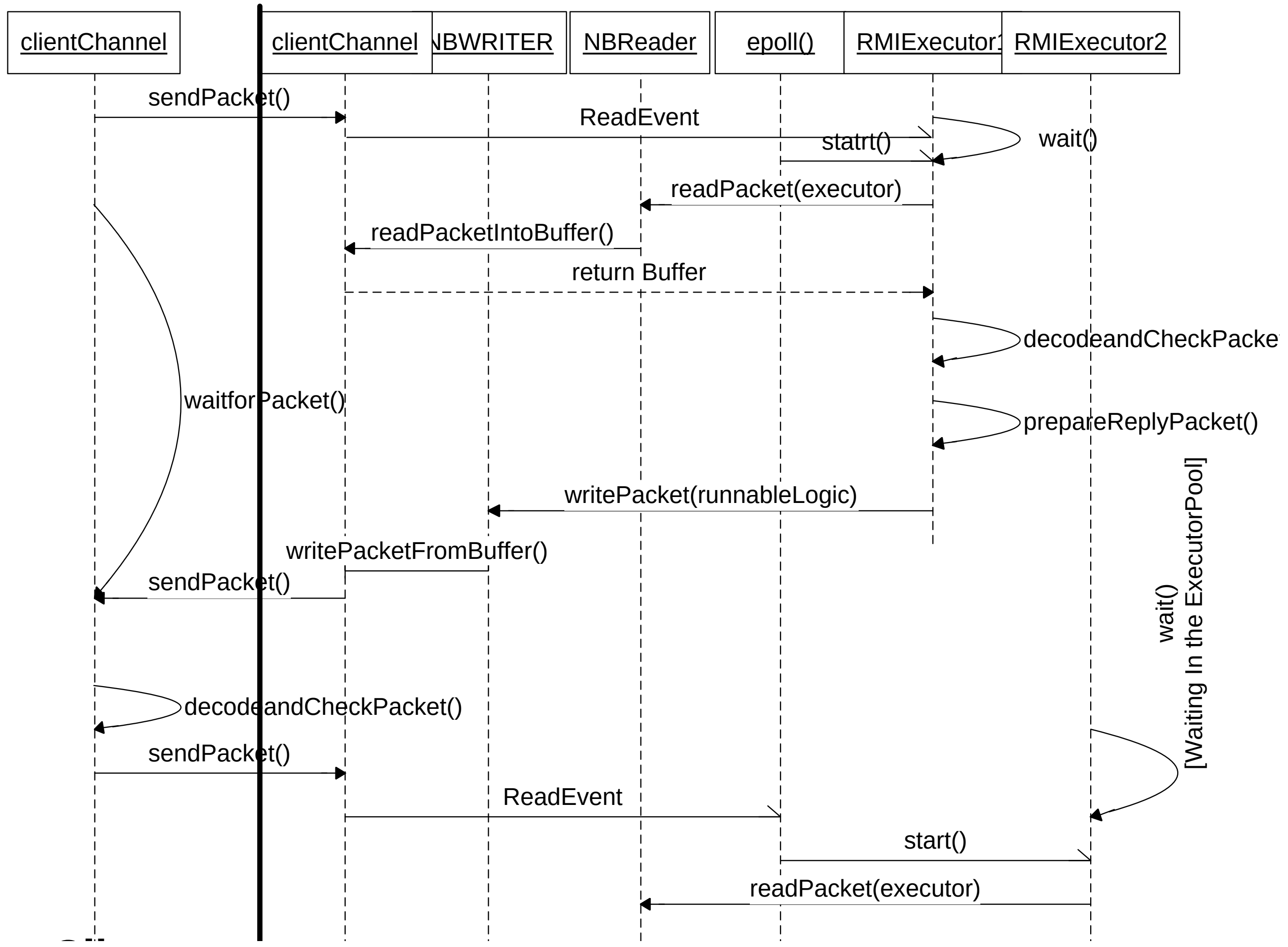


Figure 7-27 One Executor per Request Sequence

7.7 Call Handling of the Remote Method in the Stub

In our proposed real-time middleware, the invocation of the remote method is initiated on the client side of the model, and it is done through a modified stub that acts as a proxy of the remote object/component, which exists on the server machine and contains the actual remote method, where this modified stub is created using a modified version of the rmic tool from the interface that describes the remote methods offered on this remote server object as explained before. The modification of the stub was required to support our proposed implementation of the Poll Object invocation pattern within the stub. In the proposed modified model of the stub, we assume that to optimize the model for real-time systems, a Half-Asynch/Half-Synch pattern is adopted where the calling threads in the client objects that requires the execution of the remote methods on the server object(s), do their calls to the remote objects asynchronously by delivering these calls to the stub, which in turn assigns one or more of the executing threads to execute the call synchronously to the remote method on behalf of the clients, so that the call is made asynchronous by the client object's threads, but it is processed by the stub synchronously. The original model of the execution as presented by Java RMI and RMI-HRT was a synchronous model in which the calls are synchronously and directly processed by the client object's thread, which considers the stub object as a passive object that converts the local calls made by the client to remote calls over the network. Whereas, in our model, we assume that the stub is an active element that acts as an agent that processes the remote calls on behalf of the client. In the design of our proposed model of the stub, we considered the following:

- Due to the nature of the remote method call that requires creating temporary local objects locally on the client side to be used by the proxy to process the remote methods, a predictable memory model that can manage the allocation and de-allocation of these temporary objects is required. This memory model should not just manage the memory efficiently, but also it has to be user friendly in order to ease the use of the remote method calls within our model, especially with the use of the memory model of the RTSJ, which has a lot of restrictions and rules that the developer has to consider when he builds his application in this environment.
- The modified stub should support the use of our proposed communicator component. As this component can be shared among several stubs on the same container. So, the component should be attached externally to the stub object model and not embedded within it, to maximize the benefit of using the communicator component.

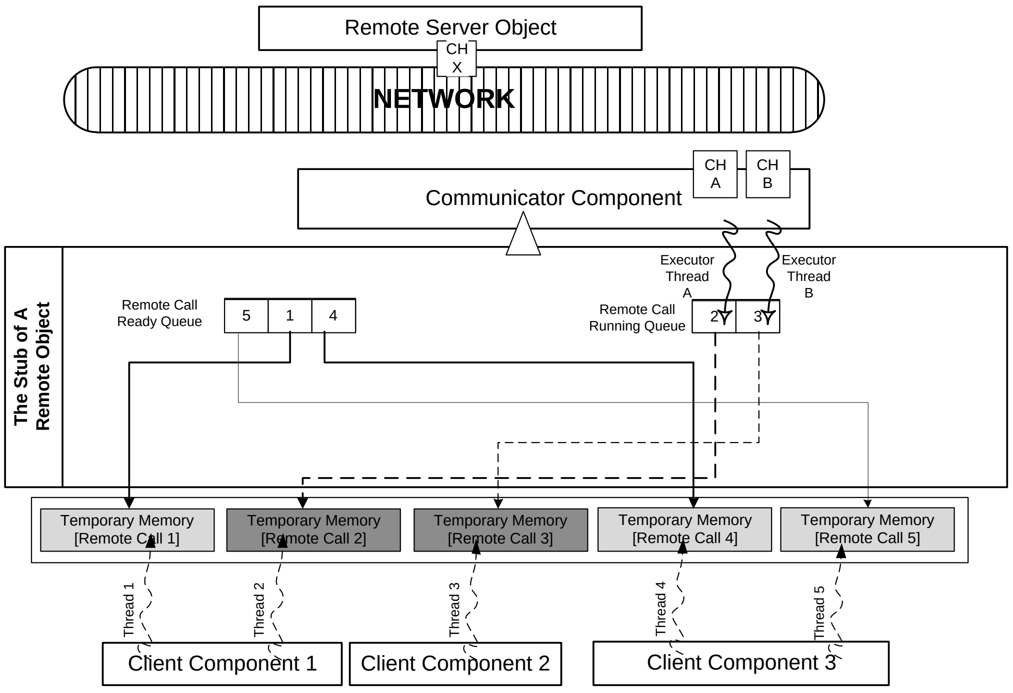


Figure 7-28 Internal Stub structure

To achieve the above requirements, we have developed the design shown in Figure 7-28, which describes the basic elements that is used for processing the method invocation in the stub. These elements are:

- **Call's Temporary Memory Areas;** we assume that each call is executed within a certain memory area, where this memory area holds and manages the temporary objects created for the call, along the lifetime of the call. In the following sections, we will present the different models of this temporary memory area.
- **A Ready Queue of the Temporary-Memories;** this queue holds all the instances of the temporary memory instances that are created by the client objects to hold the remote methods' data but have not been processed yet by the executor(s).
- **A Running Queue of the Temporary-Memories;** this queue holds all the instances of the Temporary Memory instances that the executor(s) have started executing them.
- **A reference to the communicator sub-component;** the communicator sub-component has to be attached to the stub as it represents the interface through which the stub can communicate remotely with the server objects, the communicator provides both the selectable channels that act as the end points used for the networking communication, and the executor(s) which are used for processing the calls saved in the temporary objects over the network to be executed on the remote servers, and it is responsible for receiving their results and for saving them in the temporary objects, so that the calling objects can use/check them when required.

Evaluating the Internal stub structure

As mentioned before, the stub is created within the forked memory model of the client component, so the stub can benefit from the services provided by the container to build the above structure. For example, the stub can use a specialized instance of the scoped memory life manager sub-components to create the ready queue that holds the memory. This is because the function of the queue of the scoped memory life manager is the same as the function of the ready queue, as both hold scoped memory areas that have no thread running in them. This queue can define a certain policy for getting a certain temporary memory from it, to process the call data saved within it by one of the executors; for example, using a priority queue allows the highest-priority-first policy to be used to get the call with the highest priority if the calls are defined with defined priorities.

On the other hand, the ready queue that holds the temporary scoped memory areas which are being processed, i.e. there is one executor processing the call data saved within them. So, this queue needs not to use the scoped memory life manager component, as it is guaranteed that these temporary memory areas will stay alive as there is one executor running within them, where this memory area is sent back to the ready queue, if the executor left this temporary memory area without finishing the call processing, e.g. the executor is waiting for the return value in a non-blocking mode. So, this queue acts as a passive component attached within the forked memory model of the client component, this passive component is created within the same memory area in which the temporary memory areas are created, or in a lower memory area to ensure that it can hold references to them. So, the container memory area is a guaranteed place for allocating this passive component, i.e. its CMA is the same as it ContMA, regardless of the place of allocating the stub itself.

The allocation of the temporary memory areas themselves can be from within the container memory area to ensure that they can be accessed by the executors that originate themselves from the container memory area. Also, as these temporary memory areas are used to hold the call’s data through this stub then the maximum number of these memory areas is equal to the total maximum number of concurrent calls defined for the component. Hence, as the running queue and the ready queue hold references to these memory areas only, and they hold these references exclusively, then, the maximum size of both queues is bounded by the maximum number of the concurrent calls allowed to be made through this stub, which is, in the case of a single stub/component is bounded by the total number of executors within the container. Where in the case of using multiple stubs/container the total size of all the queues of all the stubs will be bounded by the total number of executors in the container that holds all these stubs.

As the objects that hold the temporary scoped memory areas are created within the container memory area, ContMA, or any memory area accessible by the stub object and the executors, and as these objects are created dynamically with each new call made through the executor, then in order to bound their size, these objects have to be reused. So, allocating these scoped memory areas within the component has to be made from the reusable object allocator. This requires the definition of a wrapper class with *no-args* in order to wrap the scoped memory objects as the reusable object allocator sub-component accepts only classes with no-args. For these reusable memory objects, we have the following options for creating the reusable memory type/class to be used by the reusable object sub-component:

- If all the calls require identical size for their data allocated in these memory areas, then a single wrapper class is required for use by the reusable memory allocator subcomponent to encapsulate a scoped memory instance of this size.

If different method calls can be made through this stub, i.e. the general case, then we can either get the maximum memory required for these calls and use this size to create the encapsulated scoped memory area instance within the wrapper class, or if there is a limited number of calls, we can create a separate wrapper class for each with an encapsulated scoped memory area with the corresponding maximum size/call.

The above memory requirements can be used for off line analysis for calculating and checking the predictability of the memory requirements in the zero state (without logic) of the client component. But, this is not enough for calculating the worst case memory size and the memory predictability for any component used in the middleware, especially when using Java libraries within the components, because the Java libraries are built without consideration of the RTSJ, and they are assuming the use of the garbage collector for memory management. This is the case with the communicator sub-component that is used both in the client components and the server components, because this component is using the Java NIO packages. So, techniques such as code analysis of the used libraries and calculating the worst case memory sizes of the component dynamically at run time are important.

7.8 Representation of the Remote Call Data

In order to support the Poll Object pattern in our model, i.e. the non-blocking calls, we assume that several calls can be imitated concurrently from the client object's threads. Hence, to be executed by the communicator component, the parameters of these calls and their signatures and their return result have to be transferred between the caller thread and the executor thread to process them. However, it is not necessary that the execution of the calls to takes place immediately, as their might be other remote calls are being processed. Therefore, processing these calls requires these calls to be represented in a form that enable enqueuing and dequeuing them. In the same time, as these calls will be transferred over the network, it will be more efficient if they are buffered before sending them, in order to optimize the performance, Moreover, they must have additional information associated with them that identify their processing and identification requirements. Then, these requirements have to be taken in consideration when designing the format that represents the remote call's data.

In this section we discuss a proposed model of representing the remote calls, and how they can be processed. In our model, we assume that each remote method call is encapsulated and decoded within a record/packet that may has the following structure, see an example in Figure 7-29:

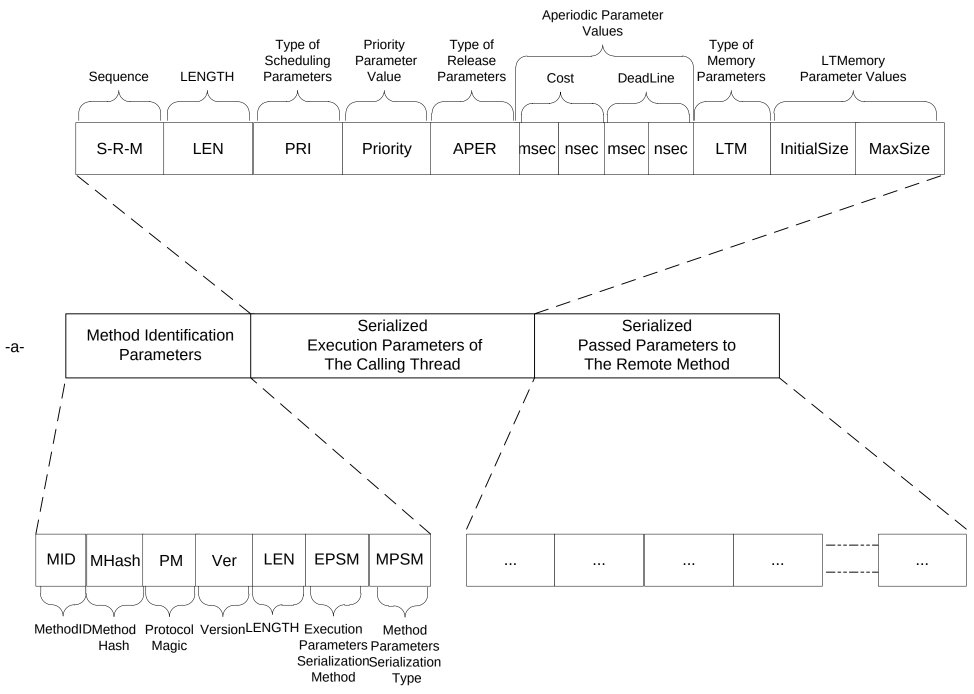
1- Method Identification Parameters. This is the essential part of the call decoded record as it identifies the called method and its structure, and how the other parts of the decoded call record are structured, and it defines how this method would be processed by the stub. Hence, we assume the following elements constitute this part:

a. Method ID. This represents the id of the method within the object.
b. Method Hash. As defined in the RMI wire protocol.
c. Protocol Magic. As defined in the RMI wire protocol.
d. Version. The version of the protocol.
e. LEN. Total Length of the packet.
f. EPSM. serialization method of the Execution Parameters in the Call Record. This can have one of the following values:
    i. [0] => No Execution Parameters.
    ii. [1] => Simplified-representation.
    iii. [2] => Normal Java Serialization.
    iv. [3] => Predictable Serialization.
g. MPSM. Method used used to serialize the Passed Parameters to the Remote method in the Call Record. This can have one of the following values:
    i. [0] => Predictable Serialization.
    ii. [1] => Normal Java Serialization.

The diagram in Figure 7-29 shows that these parameters come at the head of the call decoded record, to specify that the call is using client-propagated model, and it is using the predictable serialization method for representing the parameters.

2- Execution Parameters of the Calling Thread. The execution parameters are an optional feature within this record, and it is required only when the real-time middleware is configured to use a client-propagated execution parameters model. These parameters are represented in the record using the simplified approach that we have presented earlier in this chapter. The diagram in Figure 7-29 shows how the example previously given in this chapter for the execution parameters are located in the call decoded record.

3- Method Parameters. These are corresponding to the parameters that are passed to the remote method from the calling object. In our model, these parameters are decoded in the record structure in a serialized form, e.g. using the predictable serialization algorithm presented in (Tejera, Alonso et al. 2007) and discussed earlier in this chapter. The length of this portion is variable and, and it is even required only if the definition of the remote method assumes the existence of passed arguments. The diagram in Figure 7-29, shows how these parameters can be attached within the call decoded record.



Aperiodic Parameter Values
Sequence
LENGTH
Type of Scheduling Parameters
Priority Parameter Value
Type of Release Parameters
Cost
DeadLine
Type of Memory Parameters
LTMemory Parameter Values
S-R-M
LEN
PRI
Priority
APER
msec
nsec
msec
nsec
LTM
InitialSize
MaxSize
-a-
Method Identification Parameters
Serialized Execution Parameters of The Calling Thread
Serialized Passed Parameters to The Remote Method
MID
MHash
PM
Ver
LEN
EPSM
MPSM
...
MethodID
Method Hash
Protocol Magic
Version
LENGTH
Execution Parameters Serialization Method
Method Parameters Serialization Type

Figure 7-29 Decoded Format of the Remote Call

7.9 Example: The Future Remote Call Pattern

In order to build a stub that supports the invocation of the remote methods as future methods, it is required to extend the method invoker pattern presented in Chapter 5, to support the Future Method pattern for the remote calls. So, in this example, we present how our framework can be used to build the `FutureCallBridge` class that extends the `MethodBridge` class developed in Chapter 5, to support the operations required for this pattern. In the following, we present this class, the operations it offers, the inner elements that support these operations, and how it can be used and integerated with the client-server model developed in section 6.5.

`FutureCallBridge` **class:** This class represents the Future Call pattern; as mentioned above, this pattern has some similarities with the Method Invoker pattern, presented in section 5.8; then, to implement this pattern using our framework, we extended the `MethodBridge` class to support the specific requirements of this pattern. To provide an implementation of this pattern, we present in the following the main features that we proposed in this pattern.

1- The method call is not executed directly, but its parameters are saved in a passive component created by the caller.
2- The actual execution of the method is made by a specific handler; this handler is dedicated to execute any future call made by any component in the container enclosing this component.
3- The future passive components holding the future calls are kept alive using the Dual Fork thread memory lifetime controller running the container memory area.
4- The dual fork hold any future component until its returned object, which results from executing it by the dedicated executing handler, is no longer needed by the calling component.

From the above, we can see that there are aome modifications required in the implementation of the remote future call pattern from the method invoker pattern, these modifications are.

1- A class is required to build the passive Future component. In our example, we use the `FutureObjectComp` class for this purpose.
2- Invocation methods to suppor the internal creation of the future component, and to support the processing of the call lifetime, where this lifetime is divided into the four different operations that has to be supported by this class, these operations are:

a. Saving the call data in the future component, the saved data includes the target component name/reference, the target object, the method parameters, the memory index, and the target memory level in which it will be executed in the target component.
b. Process the call by the dedicated handler; this can be either by executing the actual required method on the target component, in case of the local call; or, in case of the remote call, by the serialization of the objects holding the saved call data, then sending them over the network connection to the target remote component.
c. Writing the Result of the call to the Future component once it is available. In the case of local methods, we assume the result object is held in the container; hence, the reference reserved in the multi named-object portal of the future component for the return object, is updated to refer to the result object. On the other case, i.e. if the call is remote; then the handler receives the returned object In a serialized form; hence, it has to deserialize these bytes first in order to get the parameters of the returned object and use it to reinitialize a reusable instance of the result object retrieved from the container's object pools.
d. The calling component can access the result object, as it is saved in the container memory area, by retrieving its reference from the future component pattern.

In the following sections, we discuss these required modifications in details; then, we present how this pattern is used in the example presented in the last chapter.

7.9.1 The FutureObjectComp class

The `FutureObjectComp` class extends the `ComponentCls` class, and it acts, as stated above, as a passive component in which the future call object and data are saved. This class has a definition of the following:

`_level:` This is used to identify the scoped memory level in which the future object is saved.

`_isResultAvailable:` This indicates if the result of the call is available or not yet.

`_caller:` This is used to specify the calling component.

`futureCompCMA`: This is a static member that identifies component memory area of this future component.

`isResultAvailable()`: An accessor method to check if the result is available or not.

`setResultAvailable()`: This method is called when the result is saved within the object, to specify that the result is ready; once the result is set ready, a notification is made to activate any thread waiting for this result.

`getResult()`: This method is used to retrieve the saved call result if it is ready; otherwise, the calling thread waits for the result to be ready, the implementation of this method is presented later in this example.

`setParameters()`: This method, shown next, is responsible of initializing the future object component, then, it adds it to the assigned container.

```
public void setParameters(final IContainer container, final String FCompName, final IComponent
caller, int level,
     final Class MemType, final int nlevels, final LTMemory initMem, final long [] InitSzs, final
long [] MaxSzs,
     final IComponent ParentComponent) {
     _level = level;   //assign the required level
     _caller = caller; //assign the caller
     init(FOBJCMA, 10000, MemType, InitSzs, MaxSzs,
        DualFork.class); //initializes the component
    setComName(FCompName);//Assign a name for the component
    container.addComponent(this); //Add the component to the container
//    FutureObjectComp.this.addTasksConfigurationClass(new ProcessCallRequest()); //
    _isResultAvailable = false; //Reset the result flag
 }
```

`attach()`: This method is responsible for attaching this future component to the given container, This requires the execution of the encapsulated method provided by the `PinFComponent` class, described next. To create a reusable instance of this class, a reference to the reusable object allocator is retrieved from the memory area of the container, then, an instance of the `PinFComponent` class is retrieved using this allocator; this instance is assigned the current future component as its parameter, and the method is executed by calling the `executeInArea()` method in the memory area of the container. After that, the stack handoff of the current future object is initialized with the memory parameters of this future objects and assigned its logic to be the one defines in this future object.

```
 public void attach(IContainer container) {
           LTMemory ContMA=(LTMemory)(container.getContainerMA());//Get the container memory area
of the caller
           IObjectAllocator allocator =
               container.getContainerObjectAllocator(); //Get the object allocator of the caller
           PinFComponent PinnerMethod=(PinFComponent)allocator.getInstance(PinFComponent.class);//
get reusable instance of the pinner method
           PinnerMethod.setParameters(this);//initialize the pinner method
  ContMA.executeInArea(PinnerMethod);//execute the pinner method in the container MA
  _stackHandOff.setParameters(LTMemory.class, 3,  new LTMemory[]{ContMA, futureCompCMA}, new
int[]{5000},  new int[]{5000},  this );//Assign parameters of the associated Stack handoff
 }
```

In addition to the above methods, the `FutureObjectComp` class includes the inner class `PinFComponent`; this class acts as an encapsulated method, where the logic of this method, which is defined in the `run()` method of this class, accesses the multi-named portal of the container of the specified future component, then it retrieves, using the reserved name `"READYFUTURECALLS",` a reference to the scoped memory lifetime controller of the ready future components from this portal, to use it as a memory pinner. Finally, this memory pinner is used to pin the component memory area of the assigned future component.

```
static public class PinFComponent implements Runnable
{
        FutureObjectComp fComp;
        public void setParameters(FutureObjectComp futureComp){
                fComp=futureComp;
                }
        public void PinFComponent(){}
        public void run(){

          final IMemoryModelControler
pinner=fComp.getCaller().getContainer().getMemoryController("READYFUTURECALLS");//Get the memory
controller of the Future Calls
          pinner.pin(fComp.futureCompCMA);//pin the memory
  }
}
```

7.9.2 The Invocation/Access Operations

To implement the four types of invocation/access operations made during the lifetime of the future component, a set of methods and classes have been defined, required for initiating the call, where in general each one of these operations involves at least the following:

1. A static front-end method within the `FutureCallBridge` class, to be used by the caller for building the future call bridge.
2. An internal method in the `FutureCallBridge` class, to initialize the parameters and the logic of the corresponding hand-off method defined for this operation in the `FutureStackHandOff` class, which extends the inner `StackHandOff` class to support the added new hand-off operations.
3. A method that executes the hand-off operation within the scoped memory stack of the future component. This method is defined in the inner class `FutureStackHandOff` class.
4. A class that implements the `IStackLogic` class to define logic of this operation, where this class is defined as an inner class of the `FutureStackHandOff` class. The assignment of a logic class to a certain operation is made using the overloaded method `FutureCall.useLogic()` defined here:

```
public void useLogic(Class LogicCls) {
   IObjectAllocator allocator =
   FutureCallBridge.this.caller.
              getMemModel().getContainerObjectAllocator(); //
   if (slogic != null)
    allocator.recycle(slogic);
   if (LogicCls.isAssignableFrom(GetResultLogic.class)) {
    slogic = this.new GetResultLogic();//we may use the allocator to create
the stack logic object to avoid memory leaks
   } else if (LogicCls.isAssignableFrom(SetResultLogic.class)) {
    slogic = this.new SetResultLogic(); //create the stack logic object
   } else if (LogicCls.isAssignableFrom(PrepareCallBytesLogic.class)) {
    slogic = this.new PrepareCallBytesLogic();//create the stack logic obj
   } else if (LogicCls.isAssignableFrom(HandOffParamsLogic.class)) {
    slogic = this.new HandOffParamsLogic(); //create the stack logic object
   } else {
    super.useLogic(LogicCls);
   }
   setStackLogic(slogic); //set the ref of stack logic in this class
  }
```

In the following, we define the methods and classes defined for each one of the four types of invocation/access operations of this pattern.

A- Initating the Future Call

The call is initiated by the caller to start the call. It involves the following elements:

1- `FutureCallBridge.makeFutureCall()`: This is a static front-end method, similar to the `MethodBridge.makeCall()` method in the method invoker pattern, this method is called by the calling component to make the call, this method starts by using the reusable objects allocator to create an instance of the `FutureMethodBridge` class, then the parameters of this instance are initialized, then the call is forwarded to the internal `_makeFutureCall()`method to start the actual call. The main difference in this method form the `makeCall()` method is that the called component is assigned as null, as the future component is created internally in the `_makeFutureCall()` method explained next. Also, as the target component is located on a remote machine, only the name of the called component is passed to this method.

```
static Object makeFutureCall(IComponent caller, String targetComponent, final Object[] callparams,
final int targetLevel,
     String FCompNAme, String tObjName) {
  IObjectAllocator allocator =
      caller.getMemModel().getContainerObjectAllocator(); //Get the object allocator of the caller
  FutureMethodBridge bridge = (FutureMethodBridge)allocator.getInstance(FutureMethodBridge.class);
  bridge.setParameters(caller, null);                          //create an object for the future
call
  Object result = bridge._makeFutureCall(callparams,targetComponent, targetLevel,FCompName,
tObjName); //calls the call internally
  return result;
 }
```

2- `FutureCallBridge._makeFutureCall()`: This method initiates the future call; it starts by getting a reference of the reusable object allocator and then uses this allocator to create an instance of the `FutureObjectComp` class, which is class for a passive component that should hold the future call data and provides operations to access it as described earlier. After it is created, the parameters of this future component are initialized, including the name of the component, the name of the target object, and the name level in which this object is saved, the calling component, and the memory parameters of this component. Then, this created component is attached to the current container, and finally, the `FutureStackHandOff` assigned to this class is assigned the `HandOffParamsLogic` to be able to process the handoff operation of the call data to the target component, then the call is delivered, using the `handoff()` method, to the future component in order to be processed.

```
Object _makeFutureCall(final Object callparams,int methodIndex, String targetCompName, final int
targetLevel,String FCompName,String tObjName) {
  final LTMemory commonMA = (LTMemory)getCaller().getMemModel().getContainerMA();//Get The container
of the caller component
  IObjectAllocator allocator = caller.getMemModel().getContainerObjectAllocator();//Get the object
allocator of the caller
  FutureObjectComponent futureComp =
      (FutureObjectComponent)allocator.getInstance(FutureObjectComponent.class); //creates an
instance of the future call Reusable Runnable Stack
  this.attach(getCaller().getContainer());//attach this future component to the container
  //so we can use a pool of precreated future objects
  futureComp.setParameters(getCaller().getContainer(), FCompName, getCaller(), level,
LTMemory.class, 3,
      commonMA, initSz, maxSz, null); //this defines exactly the target task's runnable stack
  this.targetName = tObjName;//assigns the target
  this.level = level;//assign the execution level
  getStackHandOff().useLogic(HandOffPAramsLogic);//reset the parameters
  getStackHandOff().handoff(callparams,methodIndex, targetCompName,tObjName, level  );//deliver
parameters
  return futureComp;
 }
```

3- **The** `FutureStackHandOff.handoff()` **method.** This method is responsible for initializing the parameters of the class that encapsulates the logic of the handoff operation; then it starts the execution of this defined logic in the scoped memory stack of the future component starting from the container memory area by calling the `executeInArea()` method of the container memory area.

```
public Object handoff(Object args [], int mIndex, String targetName, int
memLevel) {
       _targetName = targetName;//set the target name
       _memLevel = memLevel;//set the memory targetLevel of the obj
       backward = false;//set the handoff direction
       _methodArgs = args;//set the source
       _methodIndex = mIndex;
       _targetComp.getMemModel().getContainerMA().executeInArea(
             MethodBridge.this); //executes the call
  }
```

4- **The** `HandOffParamsLogic` **class:** This class provides the logic required to handoff the call data to the passive future component. By default, we assume that the objects of the call data are saved in the level (1) of the future component; hence, in the `runUpward()` method of this class, when the caller thread enters into the level(1), the multi named-object portal is retrieved; then the call data, including method arguments, the count of them, the index of the required method, the name of the target object, and the name of the target component are all saved as named objects within this portal, where their assigned names are reserved names to be used by the handler when they are to be retrieved for the actual execution.

```
public class HandOffParamsLogic
  implements IStackLogic {
  public HandOffParamsLogic() { }

  public void runUpWard(int curLevel, IComponent parentComponent) {
   if (curLevel == 1) { //in the specified memory targetLevel
    LTMemory curMem = (LTMemory)RealtimeThread.getCurrentMemoryArea();  //Get current memory area
    INamedObjectCollection namedPortal = (INamedObjectCollection)curMem.getPortal(); //Get the MNOP
portal
    if (namedPortal == null) {   //if there is no MNOP portal
     curMem.setPortal(new NamedObjectFastMap());  //create n MNOP portal
     namedPortal = (INamedObjectCollection)curMem.getPortal();   //Get the named portal
    }  //endif

    for (int i = 0; i < FutureMethodBridge.this.getStackHandOff()._methodArgs.length; i++) {
     namedPortal.inertObject("param_" + i,
FutureMethodBridge.this.getStackHandOff()._MethodArgs[i]);
    }
    namedPortal.insertObject("param_count",
FutureMethodBridge.this.getStackHandOff()._MethodArgs.length + "");
    namedPortal.insertObject("MethodID", FutureMethodBridge.this.getStackHandOff()._methodIndex +
"");
    namedPortal.insertObject("TargetName", FutureMethodBridge.this.getStackHandOff().targetName);
    namedPortal.insertObject("TComponent", FutureMethodBridge.this.getStackHandOff()._tComp);
  }  //end curLevel
 }   //end runUpward

 public void runDownWard(int curLevel, IComponent parentComponent) { }
}    //end logic class
```

B- The Operation of Retrieving the Result

This operation is invoked by the caller to access the result of the call, as in the Future call pattern the caller may not wait for the result, and may need to access it after doing other tasls. . The following elements are responsible for this.

1- `FutureCallBridge.GetFCallResult()`: This static method is the front end method used by the caller component; it is responsible of creating the bridge from the caller to the passive future component, and it uses this bridge to forward the call to the `_GetFCallResult()` for processing the call.

2- `FutureCallBridge._GetFCallResult()`: This method is responsible for processing the call, it specifies the `GetResuktLogic` class as the class that defines the stack logic to be executed in the future component to process the call; then, it executes the `handoffResult()` method to handoff the result from the future component to the future bridge, finally it retrieves this `Result` object from the inner `FutureStackHandOff` object of the bridge and returns it to the caller.

```
Object _GetFCallResult() {
  final LTMemory commonMA = (LTMemory)getCaller().getMemModel().getContainerMA();//Get The container
of the caller component
  IObjectAllocator allocator = caller.getMemModel().getContainerObjectAllocator();//Get the object
allocator of the caller
  this.backward=true;//assigns the direction
  getStackHandOff().useLogic(GetResultLogic.class);//reset the parameters
  getStackHandOff().handoffResult();//deliver parameters
  Object r= getStackHandOff().Result;
  return r;
 }
```

3- `FutureStackHandler.handOffResult()`: This method is used to process the hand off operation of for retrieveing the `Result` from the future component, where this logic is defined in the `GetResultLogic` class.

```
public Object handoffResult() {
        backward = false;//set the handoff direction
        _targetComp.getMemModel().getContainerMA().executeInArea(
                FutureCallBridge.this); //executes the call
  }
```

4- **The** `GetResultLogic` **class**: This class provides the logic executed by the caller component to retrieve the result from the stack of the future component. As we assume that level(1) holds the reference of the returned result object of the executed call; then, in the `runUpward()` method, once the memory level(1) is entered, the reference of the saved returned value is retrieved using the reserved name `"RESULT";` if this object is valid, then , its reference is copied to `FutureStackHandOff.Result` member in order to be usable by the calling component.

```
public class GetResultLogic
 implements IStackLogic {
 public GetResultLogic() { }
 public void runUpWard(int curLevel, IComponent parentComponent) {
  if (curLevel == 1) { //in the specified memory targetLevel
   LTMemory curMem = (LTMemory)RealtimeThread.getCurrentMemoryArea(); //Get current memory area
   INamedObjectCollection namedPortal = (INamedObjectCollection)curMem.getPortal();    //Get the
MNOP portal
   FutureMethodBridge.this.getStackHandOff().Result = namedPortal.getObject("RESULT"); //has to be
in the container
  }//end if
 }//end runUpward

 public void runDownWard(int curLevel, IComponent parentComponent) { }
}   //end logic class
```

C- Serializing the Call Parameeters

The data saved by the caller in the future component as objects; if the call is to be executed on a local method, i.e. in a component in the same container; then these objects can be retrieved by the handler and passed to the required method directly; however, in the case of a remote method, these objects have to be sent to the server for executing the remote method; therefore, these data has to be serialized first into bytes. The following elements are responsible for this.

1- `FutureCallBridge.processCalldata()`. This static method is the front end used by the handler to serialize the method data. It creates the bridge between the handler's component and the future component then forwards the processing to the _ `_processCallData()` method for the execution.

```
static Object processCalldata(IComponent caller, IComponent calledComp) {
  IObjectAllocator allocator =
      caller.getMemModel().getContainerObjectAllocator(); //Get the object allocator
of the caller
  FutureCallBridge bridge =
(FutureCall)allocator.getInstance(FutureCallBridge.class);
  bridge.setParameters(caller, calledComp); //create an object for the future call
  Object result = bridge._processCallData(); //calls the call internally
  return result; //this object has to be recycled by the caller once it finishes
using it
 }
```

2- `FutureCallBridge._processCallData()`: This method assigns the `PrepareCallBytes` class as the stack logic that defines the logic for the serialization operation, then it executes the `process()`method defined in the inner `FutureStacKHandOff` object of the bridge to process this logic.

```
 void _processCalldata() {
this.getStackHandOff().useLogic(StackHandOff.PrepareCallBytes.class);
      //reset the parameters
  this.getStackHandOff().process(); //deliver parameters
 }
}
```

3- `FutureStackHandler.process()`: this method executes the logic of the bridge to execute in the container memory area, in order to execute the stack logic defined in the `PrepareCallBytes` class.

```
public Object process() {
       _targetComp.getMemModel().getContainerMA().executeInArea(
              FutureCallBridge.this); //executes the call
  }
```

4- **The** `PrepareCallBytesLogic` **class**: The logic defined in this class is used by the dedicated handler in order to retrieve the saved call data from the future component and use them to execute the required method. In the case of the remote method, these data has to be serialized first into bytes and then it is sent over the network to the remote machine to process it. So, in the `runUpward()` method, when the dedicated handler enters the level(1) off the scoped memory stack, all the saved data are retrieved using their reserved names from the multi-named object portal of this level. After that, and when the dedicated handler enters level(2) the serialization process is applied to all the call data, where the bytes are saved into a byte buffer, which is passed to the operation that started this operation. The serialization needs to be made in level(2) to ensure that any created temporary object during the serialization is reclaimed after exiting this memory area.

```
public class PrepareCallBytesLogic
 implements IStackLogic {
 public PrepareCallBytesLogic() { }/////////////////////
 int nParams;
 String targetName;
 int methodID;
 Object [] params;
 String tCompName;
 public void runUpWard(int curLevel, IComponent parentComponent) {
  if (curLevel == 1) {
   LTMemory curMem = (LTMemory)RealtimeThread.getCurrentMemoryArea(); //
Get current memory area
   INamedObjectCollection namedPortal =
(INamedObjectCollection)curMem.getPortal(); //Get the MNOP portal
   nParams = Integer.parseIntger(namedPortal.getObject("param_count"));
   target = namedPortal.getObject("TargetName");
   methodID = Integer.parseInt(namedPortal.getObject("methodID"));
   ByteBuffer buf =
(ByteBuffer)FutureMethodBridge.this.getStackHandOff()._methodArgs[
       0]; //we can pass the buffer name in the containerMA
   tCompName = (String)namedPortal.getObject("TComponent");

   for (int i = 0; i < nParams; i++)
    params = namedPortal.getObject("param_" + i);
  }

  if (curLevel == 2) { //in the specified memory targetLevel

   Serializer.serilaize(buf, tCompName, target, params, methodID);
  }                      //end if
 }                       //end runUpward

 public void runDownWard(int curLevel, IComponent parentComponent) {
  if (curLevel == 1) { //in the specified memory targetLevel
   { }                   //end if
  }
 }                      //end logic class
```

D- Setting the Call Result into the Future Component

After executing the call, the executor needs to write the returned object of the call into the future component. The following elements are responsible for this.

1- `FutureCallBridge.SaveFCallResult()`: This static method is the static front end method used by the dedicated handler that executes the call on behalf of the caller. This method creates a reusable instance of the future method bridge between the component executed the call and the future component; then it calls the `_SaveFCallResult()` of this bridge, and passes to it an object that holds the object resulted from executing the call. In the case of the remote call, this object can be a byte buffer that holds the received serialized bytes of the received object, where it is deserialized within the future component.

```
static Object saveFCallResult(IComponent caller, IComponent calledComp,
Object Result) {
  IObjectAllocator allocator =
      caller.getMemModel().getContainerObjectAllocator(); //Get the object
allocator of the caller
  FutureCallBridge bridge =
(FutureCallBridge)allocator.getInstance(FutureCallBridge.class);
  bridge.setParameters(caller, calledComp); //create an object for the
future call
  Object result = bridge._saveCallResult(Result); //calls the call
internally
  return result; //this object has to be recycled by the caller once it
finishes using it
 }
```

2- `FutureCallBridge._SaveFCallResult`: This method assigns the required class that holds the stack logic required by this operation, i.e. the `SetResultLogic` class. Then, it does the handoff operation of the `Result` object to the future call object, and specifies the target memory level in which it is saved in the future component (e.g. we used memory level (1) in our example shown next).

```
 void _saveFCallResult(Object Result) {
this.getStackHandOff().useLogic(StackHandOff.SetResultLogic.class);
    //reset the parameters
  this.getStackHandOff().handoff(Result, 1); //deliver parameters
 }
}
```

3- `FutureStackHandler.handOffResult()`: This method processes the hand off operation required in this case, it starts by assigning the result object and the target memory level, then it executes the logic defined in the bridge, which is assigned to be an instance of the `SetResultLogic` class.

```
public Object handoffResult(Object obj, int memlevel) {
       _targetObj=obj;//set the handoff direction
       level-memlevel;
       _targetComp.getMemModel().getContainerMA().executeInArea(
              FutureCallBridge.this); //executes the call
  }
```

4- **The** `SetResultLogic` **class:** This class is called by the dedicated handler to store the returned value after executing the required method; if the call was a local call; then, this returned object has to be allocated from the object pool of in container memory area. A named-reference to this returned object is stored in the multi-named object of the future component, where it uses the reserved name "`Result`". If the call was a remote call, then the result is received as bytes from the server, and these bytes have to be desrialized first. So, in the case of the remote call, the deserialization is made when level(2), i.e. the temporary memory is entered, where the byte buffer holding these data is passed as a method parameter number(0) of the method that initiate this operation logic. Then, during the execution of the `rundownWard()` method while exiting of the scoped memory areas of the scoped memory stack, and at level (1) the reserved reference of the result is updated to refer to the returned object of the executed call.

```
 public class SetResultLogic
  implements IStackLogic {
  public SetResultLogic() { }
  public void runUpWard(int curLevel, IComponent parentComponent) {
   if (curLevel == 2) { //in the specified memory targetLevel
    FutureMethodBridge.this.getStackHandOff().Result = Serializer.deserialize(
        FutureMethodBridge.this.getStackHandOff()._methodArgs[0]); //has to be in
the container   } //end if
} //end runUpward

  public void runDownWard(int curLevel, IComponent parentComponent) {
   if (curLevel == 1) { //in the specified memory targetLevel
    LTMemory curMem = (LTMemory)RealtimeThread.getCurrentMemoryArea(); //Get current
memory area
    INamedObjectCollection namedPortal =
(INamedObjectCollection)curMem.getPortal(); //Get the MNOP portal
    namedPortal.insertObject("RESULT",
FutureMethodBridge.this.getStackHandOff().Result); //has to be in the container
} //end if
  }
 }
```

7.9.3 The Usage of the Future Call Pattern

In the example presented in the previous chapter, we developed a simple client-server application that exchange bytes packets between a client and a server component. Also, in the blocking mode of this example, the client calls were synchronous, i.e. the client has to wait for the received bytes from the client and it can not do any other processing. We can change this example using the Future Call pattern presented in this example, to enable the invocation of remote methods. To make this change, instead of sending and receiving random bytes, the output buffer of the caller has to be filled with the call data, including the serialized paramtyers, the remote method ID, the name of the target component and object. On the other hand on the server side, the Method invoker patter, presented in cghapter 5, can be used to forward the received bytes of the call to the server component to deserialize and decode the call data in order to execute the target method. In the following, we will show the key code for these operations without going in details of the serialization process itself, as it is not the aim of this example, so, we will assume the existence of a Serializer/Deserializer class that support the operations required using the methods presented in this chapter. Also, for simplicity, we avoid any synchronization requirements by assigning the same period for the communicating preriodic tasks; while instead, we set a time shift in the release of these tasks. Finally, to simplify the example, we ignore the requirements of defining the remote object and its methods, as they are not necessary for this example, as the aim of this example is to demonstrate the internal invocations made made in both the stub at the client side, and in the skeleton at the server side.

A- Using the Future Call Pattern at the Client Side

We assume that in our example, there is a component named *”Invoker”*, and *we* want to call the remote method `updateStatus()` of the object `sensor1` that resides in the memory level 1 of the remote component named “`Callee”`. Then, according to our model, then a new component object with the name “`Invoker”` has to be added to the `ClientSide` container, the logic of this component may contain the following code.

```
public static class InvokerLogic implements IStackLogic{

FutureObjectComp fcomp;
public void runUpWard(int curLevel, IComponent parentComponent){
  if(curLevel==2){
       ICloneable d1=readSensorData1();
       ICloneable d2=readSensorData1();
       fcomp= FutureCallBridge.makeFutureCall(parentComponent,
”Calee”, new Object[]{d1,d2}, 0, 1, “fUpdateComp”,“sensor1” );
  }
}
public void runDownWard(int curLevel, IComponent parentComponent){
  if(curLevel==2){
//Do not wait for the result, Do other tasks
ControlInfo cinfo=GetControlInfo();
processControlInformation(cinfo);
 }
  if(curLevel==1){
//come back to get the result of the future call
  Object result=fcomp.GetResult();
 }
}
}
```

In the above code, when the invoker enters the level (2), it initiates the future call, to deliver the data object as parameters, and specify the other data of the call as required by the method, the call returns a reference, `fcomp`, of the future component that can be used to get the result later. Then, beofre exiting the same memory level, the invoker can do other tasks, instead of waiting for the result. Finally, before exiting memory level(1), the invoker gets the results of the call using the `GetResult()` method of the `fcomp`, i.e. the future component reference.

On the handler side, the caller component, which is responsible for the execution of the call, the logic of the caller can serializ the call data into the input buffer of the caller, using a code similar to this:

```
public static class InvokerLogic implements IStackLogic{

FutureObjectComp fcomp;
public void runUpWard(int curLevel, IComponent parentComponent){
…..
……….
      FutureCallBridge.processFCallData(parentComponent,FUpdateComp,
outbuffer);

sendPacket(outbuffer);//the call bytes are sent here
…….
}
}
public void runDownWard(int curLevel, IComponent parentComponent){
……
…..
      receivePacket(inbuffer);//the returned result is received as bytes
here
      FutureCallBridge.saveFCallResult(parentComponent,FUpdateComp,
inbuffer);
 }
……..
}
}
```

In the above code, the outbuffer is passed to the `processCallMethod` to be filled with the serialized call data. On the other hand, the outbuffer that is filled with the bytes of the received serialized return object is passed to the `saveFCallResult` in order to deserialize it and save it in the future component.

B- Using the Method Invoker Pattern at the Server Side

The idea is that the data received and decoded by the acceptor handler, can be deserialized, i.e. packed into objects, in the temporary scoped memory area, i.e. level(2), of the scoped memory stack of the acceptor handler; then, the method invoker pattern developed in chapter 5 can be used to invoke a method of an object residing in the server component, this method clones these objects directly to one of the scoped memory area stack of the server component, in order to be processed by the task(s) running within the server component. The modifications of the example presented in Chapter 6 to enable this change are shown next.

The server has now a single periodic task that starts half second later than the client, assuming synchronized cocks at the server and the client sides.

```
ReusableRunnableStack servertask =
client.addPeriodicSMATask("ServerExecutor", RealtimeThread.class, null,
      new PeriodicParameters(new RelativeTime(500,0), new
RelativeTime(1000, 0)), 10000, 10000, LTMemory.class, null,
      ServerLogic.class);//Create a a periodic SMA task
task.setParameters(LTMemory.class, 3, new LTMemory[] {theContMA, theCMA},
serverinitM, maxM, server);//assign the paremeters of the task
```

In our example, we need the received objects to the server side to be passed as input parameters to the `updateStatus()`method of the `sensor1` object, where this object is allocated in the level(2) of the scoped memory stack of the server component, and the logic of the `updateStatus()`method is used to update the state of this object by the received data, and after that, it notifies the waiting thread, i.e. the handler of the server task, to process these received data.

```
public class Sensor{
  Object _data1;
  Object _data2;
  public void updateStatus(Object a, Object b){
  _data1=a;_data2=b;
  synchronized(this){
  try{
      notifyAll();
   }catch(Exception e){}
  }
 }
}
```

Hence, to implement this scenario, the logic of the Accept handler has to be changed to include this required method invocation to; this is made by first decoding the received bytes into the two objects `data1`, `data2`, then these two objects are packed in an object array, then this object array is passed to the `makeCall()`method with the other arguments that identify the required method including the names of the source(Communicator) and the target(Server) components, the target object(sensor1), and the memory level(2) in which the target object exist.

```
public void runUpWard(int curLevel, IComponent parentComponent) {
  if (curLevel == 0) { //runs in ContMA
   if (communicator == null) {
    communicator =
((CommunicatorCls)(parentComponent.getContainer().getComponents().get(
        "Communicator"))); //get ref to the communicator    }
  }//end level 0
  if (curLevel == 1) { //runs in CMA
   try {
    SelectionKey key = ((EncapsulatedHandler)getHandler()).getSelectionKey();
final SocketChannel channel = (SocketChannel)key.channel(); //get the channel
    communicator.registerationQueue.add(channel, SelectionKey.OP_READ,
        getHandler()); //register the channel
communicator.theControllerChannel.sigQueueToSignalFD(12, 10);//wake up selector
inbuf.clear();
    do {
     synchronized (getHandler()) {
      getHandler().wait();
     }                          //end synchronized
     try {
      nn += channel.read(inbuf); //read into the buffer
     } catch (Exception w) { }
    } while (nn < 40);
    inbuf.flip();
   } catch (Exception r) {
    Syustem.out.println("Exception--> " + r);
   }
  }                    //end level 1
  if (curLevel == 2) { //runs in temporary scoped memory area
   System.out.println("The Packet [" + ++PacketCount + "] has been received");
   //decode(i.e. desrialize) and process the received packet [may create
objects]
   ICloneable data1=decode1(inbuf);
   ICloneable data2=decode2(inbuf);
   Object[]params=new Object[2];
   params[0]=data1;
   params[1]=data2;
   Icomponent callee=parentComponent.getContainer().getComponents().get(
        "server");
   Icomponent caller=parentComponent.getContainer().getComponents().get(
        "Communicator");
   MethodBridge.makeCall(caller, callee, params,0, 2,"sensor1");

//.................
   }
  }                      //end level 2
 }                       //end runUpWard

 public void runDownWard(int curLevel, IComponent parentComponent) {
  if (curLevel == 2) { } //in temporary scoped memory area
  if (curLevel == 1) {   //in CMA
   outBuf.clear();
   for (int i = 0; i < 250; i++) {
    outBuf.putInt(mm.nextInt(250)); //write to the buffer
   }
        //send the packet
   outBuf.flip();
   try {
    channel.write(outBuf); //write to the channel
   } catch (Exception e) {
    System.out.println("*-----Exception----*" + e);
   }
  }

  if (curLevel == 0) { //in ContMA
   //reset the values, recycle any unrequired object ..etc.
   n = 0;  nn = 0;
  }
 }
```

On the other side of the call, the server component side, the periodic task running within this server component runs with the same period of the client, but delayed by half a second. This task retrieves in each new period the state of the sensor1 object and processes it, e.g. by display it on a monitor, or analyse the state to take a control decision. The logic for processing this scenario is shown next in the ServerLogic class which is the stack logic component of the server's periodic task.

```
 SensorData datasensor1, datasensor2;
 public void runUpWard(int curLevel, final IComponent parentComponent) {
  if (curLevel == 0) { //---=>>>runs in the container memory area
  }   //the component memory area
  if (curLevel == 1) { //---=>>>runs in the container memory area
  }   //the component memory area
  if (curLevel == 2) {//the temporary scoped memory area
       ScopedMemory mem=(ScopedMemory)RealtimeThread.getCurrentMemoryArea();
       INamedObjectCollection MNOP =
   (INamedObjectCollection)mem.getPortal(); //retrieve the portal of
       datasensor1=  (SensorData)MNOP.getObject("Sensor1");
//Processing of datasensorlis done here to ensure the deletion of any created
temporary objects when this memory level is exited
       datasensor1.makeDecision();
       datasensor1.Display();

       synchronized(this)
       {
              try(
              wait();}catch(Exception e){}}//wait for notification from the method
call made by the updateStatus() method
       }
  } //end of curlevel==2
 }  //end of runUpward

 public void runDownWard(int curLevel, IComponent parentComponent) {
  if (curLevel == 2) { }
  if (curLevel == 1) { //runs in the component memory area
  }
  if (curLevel == 0) { //runs in the container memory area
  }
 }
```

7.10 Summary

In this chapter, we aimed to prove the validity of designing and implementing the real-time middleware using the RTSJ. In order to develop this design, we developed the design through three main steps; (1) analyzing the general patterns of developing middleware solutions, (2) selecting and analyzing an open-source middleware solution as an initial model, (3) modifying this open-source model, using the patterns and component model presented in Chapter 4, and Chapter 5.

Hence, in the first part of the chapter, we presented the commonly used patterns in building the remote communication middleware software, and we discussed the collaboration among them, also we presented the different invocation patterns and how they can be integrated with the patterns of the middleware.

Then, in the second part, we described the architecture of the HRT-RMI as an open source that was built using RTSJ as a real-time middleware solution. We analyzed the structure of this architecture to identify the implementation of the different design patterns within it in order to identify the required changes that has to be made on it, in order to make enhance it.

Finally, in the third part of this chapter, we presented our own modifications and enhancements to the internal patterns within the HRT-RMI model, in order to implement our own proposed model of the RTSJ-Based real-time middleware. In these modifications, we used both the communicator component and the component framework presented earlier in this thesis, in Chapter 4, and Chapter 5 respectively. In our proposed model, we provided our own proposed structure for both the client and the server side of the middleware, associated with a description of the basic operations of the model; in addition to a description of the changes to the RMI protocol that uses the communicator component model. We then presented two models, blocking and non-blocking, that can be generated as a working model of the RMI.

A the end of this chapter, we presented an example, that shows how the method invoker pattern provided in 5.8 can be extended to enable building of Future Remote Call pattern, where this pattern integerates the communicator component developed in Chapter 6, with the memory model and its associated patterns developed in Chapter 5, to develop a method invoker and Future call invoker patterns, which can use the predictable serialization/deserialization mechanisms presented in this chapter, in order to be used for developing both the stub and the skeleton of the remote call middleware model presented in this chapter.

To evaluate the developed real-time middleware model, we provide in the next chapter an analytical evaluation for the component memory patterns and sub-components; this analytical evaluation includes first an evaluation of the characteristics and limitations of using the individual kpatterns. Also, we evaluate the effect of using the Java NIO libraries on the memory model of the communicator component, and how the Java NIO causes memory leakage within the component model, which can cause failure of the component, due to the unbounded memory consumption, then we provide our solution to this problem based on the services provided in the memory component model. Also, we evaluate the model of the stub that supports making the future calls, by analyzing the requirements of integrating the future call model with the RTSJ memory model and how our component model supports these different models to reach an optimum model for use within our real-time remote communication middleware model.

In addition, to the analytical evalutation, we provide in the next chapter, a set of expermints to measure the effieciency and predictability of the patterns and componnets that form our real-time middleware.

Chapter 8

Evaluation

There are many applications of using the middleware solutions, this made building a single middleware solution to satisfy the requirements of all these applications is an impossible task, so, as presented in, the middleware solutions can be classified into many categories according to their application and/or their structure, and there are already several well-known structural paradigms for building general middleware solutions; however, many of these technologies and paradigms can not be used directly in building real-time middleware solutions, as the original structures of most of the existing middleware technologies do not provide the basic requirement of any real-time system, which is the predictabile execution and memory management. So, as presented in Chapter 2, most of the research made for building real-mtime middleware solutions targets this issue, i.e. by modifying the structures of the existing middleware solutions, in such ways that guarantee the provision of the required predictability levels for the applications using these solutions. In addition to that, in the particular case of the Java-based middleware solution, e.g. RMI, we found that many of the unpredictability issues of these middleware solutions are inherited from the Java language itself, especially due to its inaproporiate memory management and scheduling models, as discussed in Chapter 3.

Consequently, we proposed that to build our Java-based real-time middleware, there is a need to integerate four different software technologies, these technologies are overviewed in Chapter 4, and they are:

- **The RTSJ;** to over come the limitions of the Java language, through using its new features that support real-time behavior, such as the new memory model and the scheduling models.
- **The Strategies of Networking Communication;** to support efficient communication mechanisms within our model.
- **Software Components**; to provide the reusable architectural units that are commonly used in the architectures of the common middleware solutions, where these components have to guarantee the required real-time behavior and predictability.

- **Software Design Patterns**; this technology represents the link between the previous three technologies, as we assume in our hypothesis that the structure of the Java-based real time middleware can be made using components, which are built using software patterns that use and respect the new features of the RTSJ, and in the ame time, these patterns are used to to build components that support the most efficient networking communication mechanisms for our middleware model.

Hence, we built a component framework, provided in Chapter 5 and Chapter 6, which integerated the above four technologies; then, and we used this technology to build our middleware in Chapter 7.

From the above, we see that the RSJ_based design patterns, the component model, and the communication mechanisms constitute the main parts of our middleware design; hence, these elements must be carefully selected to get a predictable behaviour of the middleware. So, in this chapter, present different levels of evaluation of our work as follows:

1- Analytical evaluation of the RTSJ-based design patterns used for developing the component model in chapter 5; this evaluation is done by comparing the structure of each pattern to the structure of similar existing RTSJ-based patterns, and by identifying the levels of memory and execution time predictability provided by these structures, as well as the level of abstraction it provides to the developer. In addition to that, we show the possible situations in which each pattern can be used, to show the level of generality it provides.
Analytical evaluation of the effects of using the Java NIO on the memory predictability of the RTSJ-based communicator component, which was presented in chapter 6, where we use code analysis technique of the basic Java NIO methods used in our model, in order to studey the effect of using the Java NIO on the memory footprint of the communicator component model, and we show how the component model provide mechanisms to overcome the problems that arise when using the Java NIO packages.
Analyitical and comparative evaluation of different possible memory models of the Future calls pattern, which was presented in chapter 7, in order to proof the optimization of the selected memory model of our proposed pattern.
Emprical Evaluation of some of the basic patterns used in developing the middleware, in order to measure their execution percformance and predictability.

8.1 Evaluation of the patterns of the component framework

In chapter 5, we have developed our component framework and model based on the RTSJ, this component model was built using a set of design patterns. So, in order to evaluate the component model, it is important to evaluate the design patterns used in building it. In this section we analyse the design patterns that we used from the following points:

1- The structure and functionality.
2- The required memory footprint.
3- The timing characteristics and/or predictable execution.
4- The level of abstraction.
5- The generality of use.

8.1.1 Evaluation of the Forked Thread, Dual Forked thread and Memory Pinner Patterns

1- The Structure and functionality

The structure of the Fork Thread pattern is using a single thread that propagates in the scoped memories that have to be held. The Dual Fork Thread on the other hand has two inner fork threads, which swap there operations to enable the addition and deletion of any of the held scoped memory areas. The memory Pinner pattern on the other hand assumes the increment of the reference count variable that controls the life time of the held scoped memory areas.

2- The required memory footprint

The footprint of both the dual fork pattern and the forkThread pattern is dependent on its implementation, for an implementation that does not reuse the runnable objects and status objects the memory footprint cost is high; whereas the memory footprint reduces significantly, when these objects are used as reusable objects. In the case of the memory pinner pattern, the memory footprint is negligible as in this pattern does not use extra objects, as it just increases the reference count.

3- The timing characteristics and/or predictable execution

For a set of scoped memories of a maximum depth of 1 level, the timing behaviour of the fork thread patterns is proportional to the number of the number of the scoped memory areas that they hold, i.e. it is of *O(n);* whereas, for dual fork thread, which has two fork threads, the timing behavior is of *O(n)* as the two fork threads are working in parallel. For the memory pinner pattern, the timing characteristics is of *O(1),* as the pinning operation is done by accessing the reference count variable of the required scoped memory area only.

In case of accessing a memory area of a depth *m* levels, the timing characteristics of the memory pinner does not change, whereas for the fork threads and dual fork thread the timing characteristics becomes of *O(nxm)*, which makes them more appropriate for use in with scoped memories of a depth of one.

4- The level of abstraction

All the three patterns have high level of abstractions, as the developer only needs to specify the required scoped memory area to be held.

5- The generality of use

These patterns are specific patterns that are used to hold the scoped memory areas, which have objects that are required to be kept alive, even without keeping any of the tasks' threads running inside it.

8.1.2 Evaluation of the Executable Logic Stack Pattern

In our component model, we have presented the stack logic pattern that enables the execution of multiple runnable objects within the forked memory model. In order to evaluate this pattern, we compare it to an older version of the pattern that we presented earlier in (Alrahmawy and Wellings 2009), and we compare both patterns as well to the normal runnable object pattern commonly used in the RTSJ. The structures of the two patterns are shown in Figure 8-1, Figure 8-2.

1- The Structure and functionality

The structure of both patterns are very close, as they both present the same function, they both consist of a set of nested scoped memory areas, where each memory area is created from its lower memory area within the stack. Also, both of them are implemented using a class that extends the runnable object; so that, they can be assigned as a single logic to be run by any schedulable object. On the other hand, the structure of the runnable object is much simpler, as it is created from a class that extends the runnable pattern, and it is normally used to execute within a single memory area.

2- The required memory footprint

The memory footprint of the two stack patterns represents one of the main differences between the two patterns. In the old version of the pattern, we assumed that the user supplies the code that executes in each memory area, either when the schedulable object enters it, or exits it as a separate runnable object. This means that for a stack runnable object that has *n* memory areas, the memory footprint of the pattern is of order *O(n)*, so this pattern needs *2n-1* runnable objects to be associated with the stack. On the other hand, with using the new version of the pattern, the memory footprint will be of *O(1)*, as there is only one runnable object assigned to be executed within this pattern. This makes the new version have more efficient memory usage.

On the other hand, as the runnable object pattern is used to run a single operation; then, the memory footprint will be of *O(1)* for this memory area only, but when used to run operations within *n* scoped memory areas, the memory footprint will be of *O(n).*

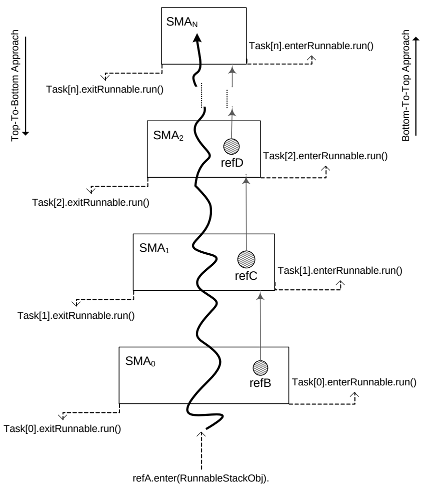

SMA_N
Task[n].enterRunnable.run()
Task[n].exitRunnable.run()
Top-To-Bottom Approach
Bottom-To-Top Approach
SMA_2
refD
Task[2].enterRunnable.run()
Task[2].exitRunnable.run()
SMA_1
refC
Task[1].enterRunnable.run()
Task[1].exitRunnable.run()
SMA_0
refB
Task[0].enterRunnable.run()
Task[0].exitRunnable.run()
refA.enter(RunnableStackObj).

Figure 8-1 The Executable Runnable Stack Pattern (V1)

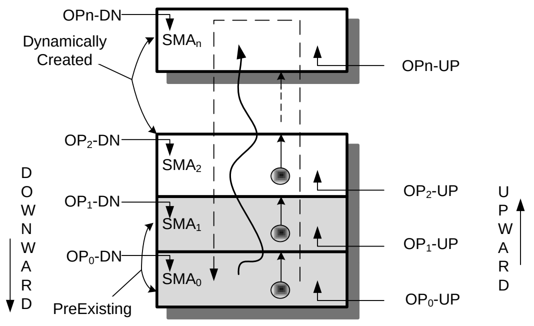


Figure 8-2 The Runnable Stack Pattern (V2)

3- The timing characteristics and/or predictable execution.

In both runnable stack patterns, we assumed that all the operations running within the runnable logic stack are running using a single thread or AEH; hence, these patterns are more suitable for running sequential operations. This in turn, makes the execution time of any task using any of these two patterns to be at least the summation of the execution times of the individual sequential operations that run within the pattern.

In the case of the runnable object pattern, this pattern is normally executed using a single executor as well to run a single operation, so its execution time depends on this single operation.

4- The level of abstraction

In the old version of the runnable stack pattern, we assumed that the developer has to present the required logic in separate runnable objects; this means that for a stack of size *n*, the developer has to present up to *2n-1* different classes within either a single file, or in separate files. In the new version of the pattern, the user has only to provide a single class that provides the logic of all the functions that represent the *entry* and *exit* operations, which simplifies the software development process and reduces the number of source code files required.

Also, in the old pattern, it was the developer's responsibility to assign each runnable object to its corresponding memory area, and he was responsible as well for linking his runnable object to the other parts of the component; however, in the new pattern, the passing of parameters that hold both the current level of memory, and the reference to the component that contains this pattern, helps the developer to manage his code in a better and more abstract way.

On the other hand, as the runnable object runs a single operation; then, it offers a low level of abstraction, as the user has to write a separate runnable object for every operation, which increases the number of required class files, and makes the code difficult to read and debug.

5- The generality of use

Due to its very basic structure, the runnable object pattern can be used in any application with the cost of the reduction of the level of abstraction. On the other hand, both runnable stack patterns provide a way for executing a sequence of nested operations within different memory areas, where the execution can run eithr from bottom to top, or from top to bottom. In our work, the two patterns in their current form are enough for our simple component model, as we have seen both patterns can be used to execute the required task's logic within the memory stack, which consists of the ContMA, the CMA, and the SMA within the component. However, both patterns in their current form do not give enough flexibility to support a wide range of applications, as they both allow the execution of a sequence of nested operations only within the stack. This means that, these stack-based patterns can be very efficient in any application requires this feature, such as the thread propagation operation to the scoped memory areas of the SMA assumed in our model. However, their flexibility is not enough to support the execution of other memory hierarchy models, in which the developer needs to use a sequence of the `enter()` and `executeInArea()` methods, e.g. the RTSJ's cactus/tree pattern. So, additional enhancements are required for these patterns to generalize their use in RTSJ applications. In our future work, we suggest a modification that can generalize the use of the new version of the pattern. In this modification we assume that, in addition to the `runUpward()` and `runDownward()` functions, the class that implements the pattern should add the `executeInLevel()` method with the following signature:

```
public void executeInLevel(int targetLevel, int operationNumber)
```

The implementation of this method should have the following pattern:

```
If (targetLevel<currentLevel)
{
      The_Target_Level = targetLevel
      NEXT_OPERATION= operationNumber;
      MA = The memory area of the level targetLevel
```

```
                nextCommand=Commands.executeInAreaCommand
                MA.executeInArea(this)
            }
```

In the above pattern, `The_Target_Level` and `NEXT_OPERATION` are two fields in the class, where `The_Target_Level` defines the required scoped memory area level, in which the operation with the index `operationNumber` has to be executed.

In the same time, the overridden `run()` method of the same class, has to be changed to add the necessary code to execute the required operation as follows:

```
public void run()
 {
      If(nextCommand==Commands.executeInAreaCommand)
       {
            runExecuteInAreaOperation(The_Target_Level,
             NEXT_OPERATION);
             return();
       }
       // code of the runnable stack pattern comes next
         …………….
            ………………..
}
```

Finally, in addition to the `runUpward()` and `runDownward()`, the class which is specified by the developer to run the logic, has to define the method `runExecuteInAreaOperation` with the following pattern;

```
public void runExecuteInAreaOperation( The_Target_Level,
   NEXT_OPERATION)

          if(The_Target_Level==0)//
          {
                 if(NEXT_OPERATION==0)
                 {
                 // the logic for a certain operation to be done
                 // using the executeInarea in the Level 0.
                 }
                 else if(NEXT_OPERATION==1)
                 {
                 // the logic for a certain operation to be done
                 // using the executeInarea in the Level 1.
                 }
                        ……………………..
          }

   }
```

8.1.3 Evaluation of the Reusable Objects Allocator Pattern

1- The structure and functionality

The structure of the object allocator pattern uses a list of linked lists, where each one of the inner member linked lists holds carriers of a certain type of reusable objects. This structure enables the management of many reusable objects by the same allocator at the same time.

2- The timing characteristics and/or predictable execution.

The use of the linked lists is not the best way for obtaining predictable timing execution. In the model, there are two levels of linked lists;

- The main linked list which holds all the types of the manageable objects. To get any reusable object, the first step is to search this types list for the required object; where, the timing of this search process is dependent on the length of the list, i.e. the number of available types, and the location of the required type within this list. In the current design, these two parameters are not well defined for the list, as there is no limit on the total number of types that can be managed by the allocator. Also, the insertion of the types in the types list has FCFS policy; this policy makes the order of the object types within the list is application-dependent. So, a better design can use a hash map to hold the types of the objects, instead of the linked list structure to get better predictable type-search process.
- The set of linked lists that hold the carriers of the objects. These lists are the second level of linked lists in this pattern. These lists do not suffer the same predictability problems of the main linked list, as they are accessed by using the last-in first-out policy, i.e. always the object on the top is retrieved or added.

3- The required memory footprint

The use of the pattern, by allowing reuse of the objects, enhances the predictability of the memory footprint of the scoped memory areas, especially in the cases where these objects are repeately created, e.g. in a loop. However, the current design of the object allocator has no restriction on the number of the objects that can coexist of any of the types. Also, it puts no restriction of the total number of the types that can be used in this allocator. So, it is the responsibility of the developer to develop his code in a way that ensures predictable maximum number of both the types and the concurrent coexisting objects of each type, in order bound the size required for this pattern to the total allocated size of the memory area from which the allocator allocates the reusable objects.

4- The level of abstraction

The level of abstraction in this pattern is high as the developer using this pattern is required to specify only either the type of the required object, or an existing object; so that, a free object similar to it is retrived from its type list by this allocator pattern.

5- The generality of use

The proposed allocator, as mentioned before, allocates objects that have a no-args constructor. This means that it cannot be used for allocating immutable objects, e.g.

Integer objects. Also, it cannot be used for allocating memory objects, e.g. LTMemory objects. So, when using this pattern, creating temporary immutable objects should be avoided and particularly in the recurrent code, and mutable ones should be used instead. In the case of memory objects, the problem is different because the memory space allocated for the LTMemory objects cannot be changed during the life time of the object. So, the best solution to reuse these memory area objects in the allocator is to encapsulate the memory objects into a wrapper class that has, in addition to a *no-args* constructor, an accessory method(s) that can be checked by the allocator, to check if the size of the encapsulated memory area object has a memory space equal to or greater than the required memory area space. For example, the allocator should have a method with the following signature:

```
Allocator.getMemoryObject(Class memType, long requiredSize)
```

This method should get a reusable area object from the types list, but instead of getting the first available memory area within the list associated with this type, a search for a memory area that has an allocated size equal or greater than the required size is made first within the linked list. This search process can be done using different policies like first-fit used, best-fit or worst-fit, with consideration of the possible delays and fragmentation problems that can arise from using these policies.

8.1.4 Evaluation of the Multi Named-Object Pattern

In order to evaluate the Multi Named-Object Portal Pattern, we will compare it with an older version which was presented in (Alrahmawy and Wellings 2009), and compare both to the the normal portal object supported in the RTSJ.

1- The structure and functionality

In the RTSJ specification, any scoped memory area can hold only a single object, the portal, which can be shared among all threads using this scoped memory area. So, to enable sharing multiple objects among threads running within a single scoped memory area, there is a need to make the portal of this memory area able to save multiple objects.

In both multi-named portal patterns, an object can be saved associated with its name within a structure that enables the retrieval of a reference of the object by its name. Then, this structure can be used as a portal of a scoped memory, to enable sharing multiple objects among the schedulable objects accessing this scoped memory area. In the older version, we assumed this structure to be a linked list; however, for the new model, we have used a hash map instead.

2- The required memory footprint

The memory footprint of any pattern is dependent on both the number of objects used within it and their sizes. In the multi-named object patterns, in both the old and the new version, the saved objects and their names are responsible for the memory footprint of this pattern. So, the memory footprint of this pattern is proportional to both the number of the saved objects that are managed by this pattern, and the size of their names, i.e. it is of order *O(n),* whereas in the case of the RTSJ portal, the memory footprint if of *O(1),* as there is only one object that can be saved as a portal object.

Another factor that affects the predictability of the footprint of the multi-named object portal pattern, and does not exist in the RTSJ portal pattern, is the size of the names of the shared objects. Although accessing the objects using their names is very flexible in many situations, this process affects the memory footprint, as the strings takes much larger size than the numerical variables, where this size is not fixed, but it increases with the increase of the length of the name; hence, a maximum length of the names have to be defined for the names. There is a trade-off between the flexibility and the memory size in the proposed component model, as we considered that the developer has to manage the size of his created objects names. However, if we need to have less memory footprint, we can change the patterns to use indexed objects instead of named objects, i.e. by identifying the objects in the hash map using numerical *ids* instead of names. This ensures better predictable memory structure.

3- The timing characteristics and/or predictable execution

The access to the RTSJ portal of a certain scoped memory area is of order *O(1),* as there is only one object assigned as a portal object. In the case of the multi-named object portal pattern, the use of linked lists in the old version makes the access to the portal's objects have no well predictable execution time, as it depends on the length of the list, this makes the worst access acces time is of order *O(n)*; while, in the newer version, the use of the hash map enhances the performance, as the average access time of using the hash maps structure is of order *O(1).* However, it keeps the worst case to be of order *O(n),* as the use of hash functions may result in collision, even with very low probability; which, if occured, requires a serach within associated linked list.

4- The level of abstraction

Both the old and the new versions of the multi named-object pattern offer a high level of abstraction, as they allow the user to use only the object name to get a reference to the saved object. This enables saving multiple shared objects in the portal instead of one, which is the case in the portal pattern in the RTSJ model.

5- The generality of use

The model is built for use in the scoped memory area specifically, to enable saving multiple objects into its single-object portal. However, the same model can be used in any memory area, to save multiple named-objects at the same time.

8.1.5 Evaluation of the Reusable Executors Pattern

1- The structure and functionality

To make a thread/schedulable object reusable, the reusable executor pattern uses wrapper class that encapsulates the schedulable object, and overrides its `run()` method to enable explicitly the management of the lifetime of the thread/schedulable object, and to add the support of changing the executed logic each time the reusable executor is reused.

2- The required memory footprint

The memory footprint of the structure of the executor itself is dependent on the assigned logic object, as it is the only mutable object within it. So, the memory footprint of the executor pattern is dependent on the footprint of the logic executing in it, which is assigned by the developer. So, it is the responsibility of the developer to manage the memory footprint, e.g. by using the reusable objects to implement the required logic. The developer has to analyze dynamically the required logic, in order to specify the memory footprint in the different memory areas ContMA, CMA, SMA, IMA; also, the developer has to specify the memory footprint with this logic, i.e. by implementing the `IForkedMemoryContract` interface as specified in the memory model.

3- The timing characteristics and/or predictable execution.

The executor pattern supports both Java threads for non-real time execution, and schedulable objects for real-time execution. By definition, the executors that use the RTSJ schedulable objects provide predictable schedulable execution, particularly when they are executing within scoped memory areas, to avoid the garbage collection interference; on contrary, the executors that use the Java threads are not predictable.

4- The level of abstraction

This pattern offers a high level of abstraction, as the developer needs only to assign the type/class that represent the required executor, i.e. either a schedulable object such as RealtimeThread, or a normal Java thread.

5- The generality of use

The reusable executor pattern supports all the schedulable objects defined in the RTSJ in addition to the Java thread, this makes this pattern applicable for any real-time application, or non-real-time application, that requires the reuse of the execution units.

8.1.6 Evaluation of the Executors Pool Pattern

1- The structure and functionality

The structure is using a single linked list for each pool, where this linked list holds all the executors managed by this pool.

2- The required memory footprint

As the pool pattern holds references to the executors, not the executors themselves, then the total size of the pool object itself is dependent only on the number of the executors it holds. Hence, as we assume in our component model that each pool has a maximum size of the executors, then the memory footprint of each pool is bounded by the size of a single executor reference multiplied by the number of maximum number of executors in it.

3- The timing characteristics and/or predictable execution.

The main operations done on the pool are the retrieval of an executor, or recycling an executor that finished its execution. The retrieval operation always takes the top element within the inner executor list, as all the free executors are identical, which makes this operation to be of order *O(1)*. Also, the recycling operation adds the recycled executor to the top of the list, which makes this operation to be of order *O(1)* as well. This makes the pool access operations are timely predicted.

4- The level of abstraction

The creation of the pool requires the user to specify the pool size, the required executor type to be in it, and the maximum required number of these executors. Then, during the initialization phase, the pool creates all these executors for the developer. On the other hand, during execution phase, the user either retrieves a free object from the pool, or recycles an executor that finished its execution. So, this pattern has high level of abstraction.

5- The generality of use

The design of this pattern makes it able to manage a list of references to the executing elements. During the initialization phase, it accepts the type of the required executor as a class, which means that it can manage either the reusable executor types defined earlier, or the non-reusable executors, e.g. Java Threads and RTSJ's schedulable objects.

8.2 Communicator Component Memory Predictability

In order to evaluate the memory predictability of the communicator component, we present here a code analysis of an implementation of the component in the jRate open source. The aim of this analysis is to identify the points that can cause memory leakage in the component memory model, and the solution that can eliminate this leakage.

JRate was built as a front-end of the GCJ, to support the RTSJ, where GCJ is a GNU VM for the Java. However, jRate has not made any changes to most of the underlying Java libraries and packages provided by the GCJ, e.g. the Java NIO package. In our model, the implementation of the communicator component is made using the epoll selector type provided by the GCJ implementation, the classes supporting this type are part of the Java NIO library. As this package is built without consideration of the RTSJ; then, the implementation of the epoll mechanism is not compatible with the RTSJ, particularly its memory model. Hence, implementing our proposed communicator component on top of this package faces a lot of problems due to this incompatibility. In this section, we cover these problems, and we present our solutions that solve these problems. In our presentation, we divide the problems into two main groups as follows:

1. **Creation Context Problems**; these problems are the problems that face the creation of the communicator component, and the difficulties of mapping it to the forked memory model.
2. **Execution Context Problems**; these problems are the problems that results from the running of the selector model and affects the memory predictability of the forked memory model.

In the following, we present the above problems and the solutions, which we proposed in our component model to solve these problems.

8.2.1 Creation Context Problems

These problems are mainly due to the fact that many of the Java NIO classes used by the selector model are using static fields. As defined in the RTSJ, the use of the static fields requires their allocation to be done in the immortal memory. This means that the developer is enforced to do the allocation in the immortal memory, and he is inhibited from mapping these objects to any of the different memory areas of the forked memory model. This is not recommended in our proposed component model, as at the termination time of the component, it should not leave any memory behind it, to not affect the other parts of the application. Moreover, objects created in the immortal memory are not allowed to access any other objects in any scoped memory area, which may enforce the developer to allocate many of the objects, if not all, in the immortal memory area. This problem affect the predictability memory model of the application significantly, especially if the creation of these objects is repeated frequently, as this will lead to memory leakage in the immortal memory area, and even may lead to system failure, as the immortal memory may reach a point when it becomes completely filled and cannot accept any more objects, unless a reusable object mechanism, such as the one provided in our component model, is used to keep the required memory size bounded.

To avoid these problems, it was necessary to analyze the creation process of the objects constituting the model, to ensure that the main objects are allocated either in the current memory area, or in the user specified memory area, instead of allocating them in the immortal memory area.

We have found that the selector object itself has to be created itself in the immortal memory area, if the normal creation procedure is followed. The creation procedure is done in the `CommunicatorCls` class, and it includes the following steps:

1. The reference of the selector is defined as a field in the class

```
Selector selector;
```

2. The selector object is created in the container's ContMA

```
selector=EpollSelectorImpl.open();
```

The above steps show the creation of the selector, which is the main element of the communicator component model that is responsible for monitoring the events occurring on the registered channels. The creation is done using the static method `EpollSelectorImpl.open()`, this method is inherited internally from the parent class Selector, where the `Selector.open()` method executes the following statement:

```
SelectorProvider.provider().openSelector();
```

The above statement runs the `openSelector()` method on the provider retrieved by the `provider()` method, where the call of the `openSelector()` creates the selector in the current memory area when it runs the statement:

```
return new EpollSelectorImpl(this);
```

The purpose of the `provider()` method is to return the object that represent the system default provider, which is used to create the selector object. This provider object is used as well each time a new channel is created within the component, whether it is a server channel or a client channel, as the provider is responsible for creating the channels, where the normal creation of the channels is done for a client channel using the statement:

```
SocketChannel theClientChannel=SocketChannel.open();
```

or, for a server channel that uses the statement:

```
ServerSocketChannel theServerchannel=ServerSocketChannel.open();
```

The main purpose of the `provider()` method is to return the system default provider, which is defined as a static field `systemDefaultProvider`. This provider is either created as an instance from the `SelectroProviderImpl` class, or it is dynamically loaded at design time from the value of the property "`java.nio.channels.spi.SelectorProvider`". In both cases, when the `provider()` method is called for the first time, a new instance of the `SelectroProviderImpl` is created in the current memory area or the user specified area, and this created object has to be assigned to the static field `systemDefaultProvider`; however, this assignment, if the created object's memory area is a scoped memory area, is not allowed in the RTSJ, and it will raise an exception as objects in scoped memory areas have shorter lifetimes than objects in the immortal memory area. Hence, in this case, the developer is enforced to create the selector object either in the immortal memory area, or the heap memory area, which is not recommended, as it will enforce most of the other objects in the communicator component to be allocated in the immortal or heap memory area, and it may make the forked memory model useless, unless we defined the container itself to be created in the immortal memory area as well.

To solve the above problem, we have assumed that the creation process of the selector is done in non-abstracted form, i.e. the creation process is detailed in well-defined multi-steps that avoid the use of hidden static objects.

So, in our implementation we use the following steps:

1- A private field is defined as follows in the `CommunicatorCls`:
```
Selector ProviderImpl provider;
```

2- The provider is created in the current memory area during the initialization phase using:

```
Provider=new SelectorProviderImpl();
```

3- The selector is created as well in the current memory area using

```
selector=new SelectorProviderImpl(provider);
```

4- The selector is opened using

```
Provider.openSelector();
```

At the same time, we can change the creation of the server channels and the client channels to use the following forms:

```
SocketChannel theClientchannel=
                  provider.openSocketchannel(), and
ServerSocketChannel theClientchannel=
                  provider.openServerSocketchannel();
```

The above statements create all the required objects in the same memory area, to ensure that the developer has the full control of choosing the allocation memory area that suits his model. For example, in our proposed model for real-time middleware, we can choose the container memory area, ContMA as the memory area in which these objects are created, to ensure that they are all available to all components and sub-components within this container.

8.2.2 Execution Context Problems

The execution context problems are those problems that may arise, due to improper use of the memory model during the execution phase of the component. In addition to the illegal memory assignments, these problems include the unbound use of memory areas of the forked memory model. These problems can occur easily in the implementation of the component; if the original Java NIO libraries are used. This is because the original Java NIO libraries are built assuming the automatic memory management, i.e. by using the Java's garbage collector, which is not the case when we use the immortal or scoped memory areas to build all the parts of the forked memory model of the communicator component. So, it is very important to analyze the code of the Java NIO libraries used for building the communication component, to identify the problems that can arise when it is used within the forked memory model of the communicator component. Most of the problems are related to the selection keys life cycle memory model. In the following, we present these operations and the main problems that we have spotted in them; then, we present our solutions to these problems based on our component model.

In the open-source files of jRate, most of the selection key management operations are associated with the epoll selector's implementation within the EpollselectorImpl.java file. In the following, we summarize these operations, and introduce the memory management problems that arise when using this implementation to build our communicator component. Then, we present our solutions to these problems.

A- Selection key registration operation

Selectable channels, whether they are server or client channels, register themselves with the selector with a set of registered events, to observe when these events occur on them. The registration of a selection key is made using the method `AbstractSelectableChannel.register(Selector, events, attachment)`. This registration process starts by searching if the channel already has a key registered with the selector or not, where the search is done in a hash map associated with this channel, which holds the keys defined for this channel with all the selectors available in the system, as the Java NIO assumes that the channel may be registered with more than one selector. The result of the search is as follows:

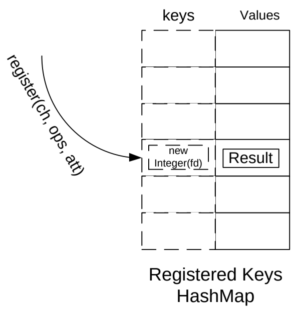


Figure 8-3 registering a selection key

1. If the key is found, the found key is updated with the new events and attached object, and the registration process finishes.
2. If the key is not found, i.e. the channel was not registered; then, the registration process is forwarded to the selector object itself, i.e. by calling

   `selector.register(this, events, attachment)`,

   which is defined in the *EpollselectorImpl.java* file.
3. In the `selector.register()` method, a check is made, before the registration process, on a hash map that holds all the keys associated with this selector, see Figure 8-3, to check if the key is already registered or no. This is done using the statement:

   `keys.containsKey(new Integer(native_fd))`
4. If the key is found, i.e. the channel is registered; then, an exception is thrown,

5. otherwise, the key is not found, the registration is made of a new *result* object that holds the information of the registered key,

```
EpollSelectionKeyImpl result= new
      EpollSelectionKeyImpl(this, channel, native_fd);
keys.put(new Integer(native_fd),  result);
```

where, *result* is the object that holds the registered information, including the events, attached object, etc.

Memory Analysis and Problems

The channel's associated linked list holds the selectors with which this channel is registered, so its size is bounded by the number of the selectors that we can attach this channel to. In our component, we use a single selector; this ensures that size of the linked list is bounded to a maximum size of one, as long as the channel is not used by any other selector.

During the execution of the `selector.register()` method, two immutable Integer objects are created, to be used as keys within the hash map, the first one is used for searching within the hash map, while the other one is used for adding a new entry to the hash map. Also, the *result* object is created and inserted in the hash map. From RTSJ perspective, these objects are created in the current memory area each time the registration is made for a new channel. This makes the memory allocated by the `register()` method is unbounded, which result in memory leakage.

B- Selection Operation

The selection operation `Selector.select()` is the main operation in our component, as this operation makes the selector to start/continue observing the events occurring on the channels registered with it, and save the information of these events in the `selectedKeys` hash set, see Figure 8-4.

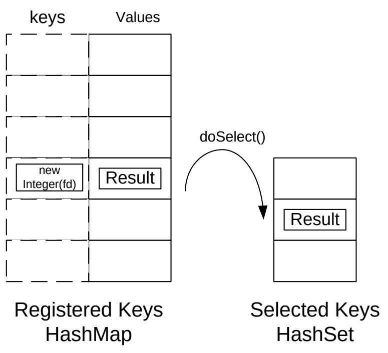


Figure 8-4 Selection-Operation

This operation executes the `Selecor.doSelect(-1)` method, which causes the executing thread to wait indefinitely until one of the registered events occur. The `Selector.doSelect()` operation involves three main tasks as follows:

1. Cancelled keys handling.

Before waiting for the registered events, the selector first needs to check if any of the registered keys have been cancelled, to avoid monitoring the events belonging to it. In the Java NIO's Selector model, the Selector has a hash set that holds all the cancelled keys, any key cancelled, i.e. using `key.cancel()`, is added to this hash set, see Figure 8-5. In the `Selector.doSelect()` method, the elements of this hash set are retrieved, and the following operations are done on every cancelled key in this hash set:

a- Its file descriptor is deleted from the epoll's associated file descriptors.

b- The cancelled key is removed from the hash map of the registered keys using:

```
keys.remove(new Integer(key.fd))
```

c- It is removed from the `cancelledkeys` hash set and deregistered.

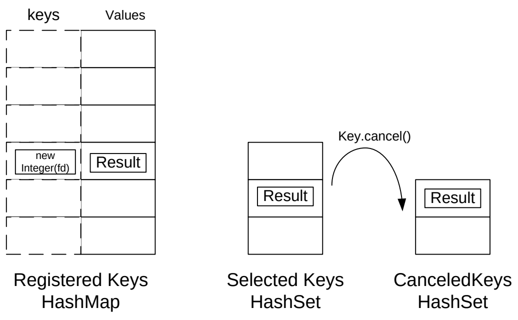


Figure 8-5 Cancelling a key

2. Clear out the closed channels

Before waiting for the registered events, the selector needs to check if any of the registered channels have been closed, in order to remove it from the hash map of the registered keys. So, a check is made on all the keys entries of the registered keys hash map, to check if any of the channels associated with these keys are closed, if found, the key associated with this closed channel is removed from the registered keys hash map.

3. Generated events handling

After receiving events on the channels registered with the selector, the selector generates a hash set that hold keys for all the observed events, see Figure 8-6. This process is made in the `Selector.doSelect() method,` by first creating a new hash set of the size *ret*, which represents the count of the observed events, where this hash set is used to hold the selected keys. This operation is done using the statement:

```
HashSet S=new HashSet(ret)
```

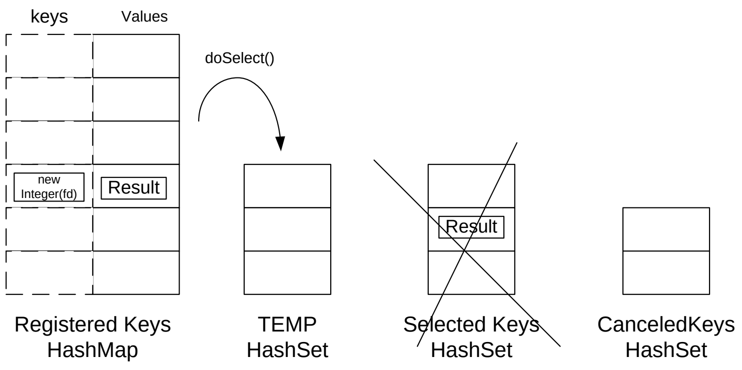


Figure 8-6 Creating a s new selected keys hash set

Then, the following sequence of operations is repeated *ret* times, for each one of the observed events monitored by the selector:

1. Get the position of the next epoll_event in the events byte buffer that is filled by the epoll.
2. Generate a new byte buffer to hold the information of this event using

   ```
   ByteBuffer b= events.slice()
   ```

3. Get the file descriptor associated with this event using:

   ```
   fd=selected_fd(b)
   ```

4. Get the key corresponding file descriptor `fd`, and update the information of this event using:

   ```
   EpollSelectionKeyImpl key =
       (EpollSelectionKeyImpl) keys.get(new Integer(fd))
   ```

5. Update the key values by the events data.
6. Add the key to the hash set S.
7. Reallocate the events buffer by using the method `reallocateBuffer()`, which uses the method `ByteBuffer.allocateDirect()` to regenerate the events buffer.
8. Assign S to `Selector.SelectedKeys`.

Memory Analysis and Problems

The creation of the hash set S temporarily is acceptable when the allocation is in the heap, as the garbage collector can deallocate it, when it is not in use; however, the allocation of it in a scoped memory or immortal memory would make memory leak, because this hash set is created frequently during the lifecycle of the component.

Similarly, the generation of the byte buffer `b` in the scoped memory area generates objects at run time, where these objects cannot be deallocated.

Also, getting the key from the hash map involves the creation of the immutable Integer object, and this process is repeated in all iterations, and at each time a new event(s) occurs, and is observed by the selector.

Moreover, the reallocation of the events buffer using the method `reallocateBuffer(),` is one more method that makes memory allocation unbounded.

Finally, assigning a new created instance of S to `Selector.SelectedKeys` means that the current object assigned to `Selector.SelectedKeys` will be not in use, and as this object is created out of the heap, so, there is no dynamic memory management to reclaim this object; hence, this object would causes a memory leak.

8.2.3 Solutions to the Memory Problems

All the above memory problems are due to the fact that the Java NIO packages were built using the standard Java that supports dynamic memory management, so if we used the heap memory area for the allocation of the selector object, all these problems would be eliminated, but with the cost of the need to an efficient and predictable real-time garbage collector. However when using the scoped memory area, or the immortal memory area, the above problems occur. Most of these problems are due to the fact of the creation of temporary or hidden objects within the Java NIO packages in a scoped/immortal memory area. In our component model, we proposed two solutions to these problems:

1- The use of reusable objects using the reusable object allocator sub-component/pattern.
2- The creation of the these temporary objects in nested scoped memory area, then exit from it directly once the temporary object(s) are not in use any more.

The use of the second solution to solve the above problems is not useful with the above problems, as all the operations that involve the creation of the hidden/temporary objects are based on the selector object, basically its `selector.select()` operations, where this selector object has to be alive for the lifetime of the component, as the `selector.select()` method has to be executing, as long as the component is alive, to monitor the network I/O events. For example, the creation of `new Integer(fd)` in a nested scoped memory area, and using it to be a key for the selection keys hash map would not be possible, as the hash map, which is a field of the selector, would be created in the scoped or immortal memory area in which the selector object is created, i.e. using the temporary Integer object, however, this creation process is not acceptable in the RTSJ, as the hash map has a shorter life time than the selector object, and accessing it from the selector object violates the single parent rule.

On the other hand, the first solution is more general, and applicable within any memory area; however, it has the restriction that it requires the reusable objects to be created from classes that have *no_args* constructor only, i.e. it cannot be used for immutable objects like Integer object.

Hence, as the first solution is not feasible, we adopted the use of the second solution, i.e. using the reusable object pattern. In order to use the reusable object pattern, we have to avoid the restriction of using the immutable objects; so, we have to use mutable object instead of the Integer immutable object. The following code shows a simple, class which replaces the immutable `Integer` class.

```
public class MutableInteger
{
        public int val;
        public MutableInteger()
        {//no args constructor
        }
        public void setValue(int v)
        {
                val=v;
        }
        public void getValue()
        {
                return val;
        }
        public int hashCode()
        {
                return val;
        }
        public boolean equals(Object other)
        {
                 if (this == other) return true;
                 if     (!(other     instanceof     MutableInteger))
                       return false;
                 MutableInteger otherMutableInteger =
                         (MutableInteger) other;
                 return
        MutableInteger.val==otherMutableInteger.val;
        }
}
```

The above class has only an integer field to hold the file descriptor as a key within the selection keys' hash map. The class has to override the `hashCode()` and `equals()` methods, as they are used by the hash map internally. The `hashCode()` method is overridden; so that, it returns the value of the integer field as the hash code. This ensures that all objects assigned the same integer will be considered the same object, so that they can be used to access the same entry in the hash map. The `equals()` method ensures that the equality of two objects of the same class is dependent on their state not behaviour, i.e. as long as they hold the same integer value, they are considered the same.

In order to use the reusable object allocator, the following sequence of steps has to be made:

1. Get a reference `objAllocator` to the object allocator sub-component, this is done using the `MNPORTAL` of the current memory area, where we assume that every scoped memory area within the forked memory model has a reference to the reusable memory object allocator; either local to this scoped memory area, or global in the container that holds the current component.
2. Get an instance of the mutableInteger class.
3. Assign the required file descriptor to this mutableInteger.
4. Use this object in place of the `new Intger(fd)`.
5. Once it is not in use, recycle it.

The following code snippet shows the changes within the `register()` method:

```
int native_fd=channel.getState().getNativeFD();
synchronized (keys)
{
       MNPORTAL  mp=( MNPORTAL)
       MemoryArea.getCurrentMemoryArea().getPortal();
       IObjectAllocator allocator=( IObjectAllocator )
              mp.getObject("ObjectAllocator");
       MutableInteger  mutInt=
              allocator.getInstance(MutableInteger.class);
       mutInt.setValue(native_fd);
       if(keys.containsKey(mutInt))
              throw ……
       EpollSelectionKeyImpl result= (EpollSelectionKeyImpl)
       allocator.getInstance(EpollSelectionKeyImpl.class);
       result.setParameters(this, channel, native_fd);

       ……………
       …………….
       result.selectedOps=0
       ………………
       keys.put(mutInt, result);
       ………..
       ………
       allocator.recycle(mutInt);
}
```

In the above code, we must note that we have used the same object mutInt for checking the hash map; then, for adding the key, we need to create another instance, because it holds the same file descriptor. Also, we have to note that we have recycled the object at the end of the function, as the hash map does not keep an instance of the object, because it uses its hash code only as an index within the bucket of the hash map.

In addition to that, we see that we have used the allocator to get an instance of the `EpollSelectionKeyImpl` class, but we have not recycled it. This is because this instance will reside in the hash map, untill it is cancelled by the user, or the channel is closed; so, it cannot be recycled here. Also, we have to note that, we used the `setParameters()` method, this method is an addition to the `EpollSelectionKeyImpl` class to enable reusing it.

In the `doSelect()` method, another set of changes have to be made, as it contains a set of problems other than the temporary objects, in the following we present the changes made to the main parts of this method.

1. Cancelled keys handling.

The problems in this part are regarding the temporary/hidden object, the changed code of this part is as following:

```
for(Iterator it=cancelledKeys.iterator();it.hasNext();)
{
       MNPORTAL  mp=(MNPORTAL)
                    MemoryArea.getCurrentMemoryArea().getPortal();
       IObjectAllocator allocator=mp.getObject("ObjectAllocator");
       EpollSelectionKeyImp key = (EpollSelectionKeyImp) it.next();
       epoll_delete(epoll_fd, key.fd);
       key.valid=false;
       MutableInteger  mutInt=
              allocator.getInstance(MutableInteger.class);
       mutInt.setValue(key.fd);
       EpollSelectionKeyImp  oldObject=keys.remove(mutInt);
       It.remove();deregister(key);
       allocator.recycle(mutInt);
       allocator.recycle(oldObject);
}
```

The first main change in this part is the use of the reusable object mutInt; then, recycling it at the end as explained before. The other change is the addition of the `allocator.recycle(oldObject),` to recycle the cancelled selection key as it has no more use, this key was generated during the `register()` operation.

2. Clear out the closed channels

The problem in this part is that the keys of any closed channels are removed from the keys hash map, and once they are removed, they are not used any more;

hence, the solution for this part is to recycle these keys. The changes to this part is shown in the next code snippet.

```
for(Iterator it=keys.values();it.hasNext();)
{
       MNPORTAL  mp=(MNPORTAL)
       MemoryArea.getCurrentMemoryArea().getPortal();
       IObjectAllocator allocator=(IObjectAllocator)
       mp.getObject("ObjectAllocator");
       EpollSelectionKeyImp key =
               (EpollSelectionKeyImp) it.next();
       Selectablechannel ch=key.channel();
       if(!((VMChannelOwner)ch).getVMChannel().
              getState().isValid())
       {
              it.remove();
              allocator.recycle(key);
       }
}
```

3. Generated events handling

The problems in this part are different from the previous parts, and each one of these problem needs a different solution. We present the solutions of these problems next.

a. **The temporary hash set** S

In the original code of the `doSelect()` method, a new `selectedKeys` hash set instance S, is created after each selection process, where the size of this hash set size is assigned to be of a size equal to the number of the events observed by the epoll selector. Then, this new instance replaces the old `selectedKeys` hash set. Our solution assumes that we can avoid using this hash set completely, by directly reusing the original `selectedKeys` hash set, as the aim of using the temporary hash set S is to fill the `selectedKeys` hash set. This helps as well to avoid the need for cleaning or reusing the discarded old `selectedKeys` hash set. But to do this, we have to ensure that we discarded all the old elements of the `selectedKeys` hash set first, before refilling it with the new events entries. The remove process should recycle the removed objects. In the same time, as the `selectedKeys` hash set is not recyclable; then, the `selectedKeys's` size should be set to have a fixed size, which has to be at least equals to the meaximum number of conocurrent registered network I/O events observed by the associated selector object. Hence, care must be taken to avoid the use of the `selectedKeys.size()` method in any other method, as it would get the maximum possible number, instead of the actual number, of the observed events. So, a separate integer field in the selector class can be provided to hold the current actual number of elements within the `selectedKeys` hash set.

In our component model, the size of the `selectedKeys` hash can be bounded, as we proposed in the model that we limit the concurrency to a certain limit; so, by analyzing the total number of concurrent schedulable objects, and by analyzing their code, we can determine the total number of channels registered concurrently in the selector, and the maximum number of events for each channel; hence, we can know the worst case of the memory size required for the `selectedKeys` hash set.

b. **Generates a new byte buffer for the event and get the file descriptor from it**

The events byte buffer holds the bytes of all the events observed by the selector, in order to decode the data of each event such as the file descriptor, fd, associated with this event.The operation `ByteBuffer b=events.slice()` generates a new direct byte buffer, `b,` in each iteration, this byte buffer holds the data bytes of the events that have not been decoded yet. The `ByteBuffer.slice()` method is called to generates a new direct byte buffer from the original events buffer, where the bytes of this buffer are read from within the events buffer, `events`, starting from the current position, where this position is updated in each iteration, to start form the bytes of the next event(s) in it that have not been read yet. Then, the generated byte buffer `b` is passed to the method. `selected_fd(),` which decodes the bytes of this buffer, and works on the bytes of the first event in it only, to get the file descriptor associated with it, and ignores the bytes of other events. So, this means that for *n* of observed events, a number *n* of non-reusable byte buffers objects are created, and this process is repeated in each selection operation, which would cuse memo leakage. We can avoid this problem by avoiding the use of `events.slice()` completely, as it is the source of the problem, and change the logic of the `selected(fd)` method in a way that makes it work on the original events byte buffer instead of the generated one, but this requires that the order of the required event to passed to it each time it is called, in order to be able to identify and decode the next event in sequence.

c. **Using new Integer(fd)**

The solution of the creation of a temporary Integer object using `new Integer(fd)` is the same as mentioned before.

d. **The reallocation of the events buffer using** `reallocateBuffer()`

In the original code of the `doSelect()` method, the execution of the `reallocateBuffer()` is done, to dynamically allocate a new direct byte buffer that has a size equal to the number of the currently occurred events. This behaviour is acceptable when we allocate it in the heap memory area, as the garbage collector can free the memory of the discarded events byte buffer each time a new one is generated, but in case of the allocation of the events byte buffer in a scoped memory area, this is not acceptable, exactly as it was the case with the temporary hash set S mentioned before. So, to solve this problem, we create the hash set only once with its worst case maximum size, to avoid the reallocation of the buffer completely, where the size of this byte buffer has to be bounded. So, again, as it was a requirement for the `selectedKeys` hash set, we need to limit the level of concurrency in the component, in order to be able to analyze the system in order to identify and limit the worst case of the maximum number of the concurrent occurring events in the component. So that, the size of the events byte buffer can bebounded.

According to the above solutions, we can change the code of the handling of the generated events to the following:

```
MNPORTAL  mp=(MNPORTAL)
MemoryArea.getCurrentMemoryArea().getPortal();
IObjectAllocator allocator=mp.getObject("ObjectAllocator");
for(Iterator it=SelectedKeys.iterator();it.hasNext();)
{//this part to recycle the old selected keys
    EpollSelectionKetImpl oldkey=
        (EpollSelectionKetImpl) it.next();
    it.remove();
    allocateor.recycle(oldkey);
}
for(int i=0;i<ret;i++)//ret is the number of observed events
        {//this part is to encode the events to generate the new
   // selected keys and add them to the selectedKeys hash set
           events.position(i*sizeof_struct_epoll_event);
           MutableInteger  mutInt=
           allocator.getInstance(MutableInteger.class);
        //get the fd of the event [i] in the events bytebuffer
           int fd=selected_fd(events,i);
           mutInt.setValue(fd);
           EpollSelectionKetImpl key =
           (EpollSelectionKetImpl)  keys.get(mutInt);
        ………
        …………
           selectedKeys.add(key);
}
```

8.3 Evaluating the memory model of the future calls

In our real-time middleware model, we assume that any invocation of a remote method made by one of the client component's threads is delivered to an executor of the sub-communicator component, where this executor has to process the call on behalf of the client component's thread; so that, the client object/component's thread becomes able to continue doing other work, instead of waiting idle during the execution of the remote method. Hence, the data associated with the call, such as the method name and the parameters to be passed to the remote method for execution, has to be shared between the calling thread and the component's executors, where these data are temporary data that have to be either reclaimed, or reused after the end of the call execution. According to this, a shared memory model that is compatible with the RTSJ memory model has to be provided in any RTSJ-based real-time middleware model, to support the proposed invocation patterns.

To our knowledge, this model has not been used in any RTSJ based model, and only the synchronous model was used. For example, in the RMI-QoS (Tejera, Tolosa et al. 2005), as presented in chapter 3, the authors assumed the clients are able to make only synchronous calls, and there was no support for the future calls, so they assumed a simple memory model for the client side, where the default allocation context of the invoking thread is used to allocate the object that returns as a result of the remote invocation.

So, in this section, we evaluate a set of proposed different memory models that can be developed for sharing these data within any RTSJ-based middleware models by analyzing these models and comparing them according to:

1. The scenarios of use of each pattern, to proof the validity of the model.
2. The level of btsraction, by identifying the required steps to access the future object.
3. The validity of supporting different invocation pattern(s).
4. The restrictions of use.
5. The validity and support of the model in the forked memory model.

In our analysis, we present the scenarios of each model first, independent of the forked memory model; then, we show how this model can be supported in the forked memory model.

A- Immediate Nesting in the Stack of the Calling Thread

Scenarios: In this model, the calling object/component’s thread creates a new scoped memory area, which is nested just on top of this thread’s current memory area, to hold the data of the remote call. The block diagram, shown in Figure 8-7, shows two scenarios of this model; where there are two different real-time threads are executing these two scenarios to process the remote calls. In the first scenario, we assume that the first thread is initially running in the top memory area C, when it intends to make the call. In this scenario, in order to make the remote call, the first thread creates a nested scoped memory area D and enters it to make the call from within it; this memory area is created in order to hold the temporary parameters of the calls during making the call, and must be reclaimed once the call finishes execution. The executor of the communicator component must be able to do the memory entries A=>B=>C=>D to process the call.

In the second scenario, the other thread is supposed to be running in memory area F in the middle of the scope stack, i.e. using the `executeInArea()` call, at the time that it wants to initiate the remote call. Then, according to this model, the thread enters the temporary scoped memory area, I, in order to run the call, and the executor, similarly to the first case, has to be able to enter the scoped memory areas E=>F=>I in order to access the call's data.

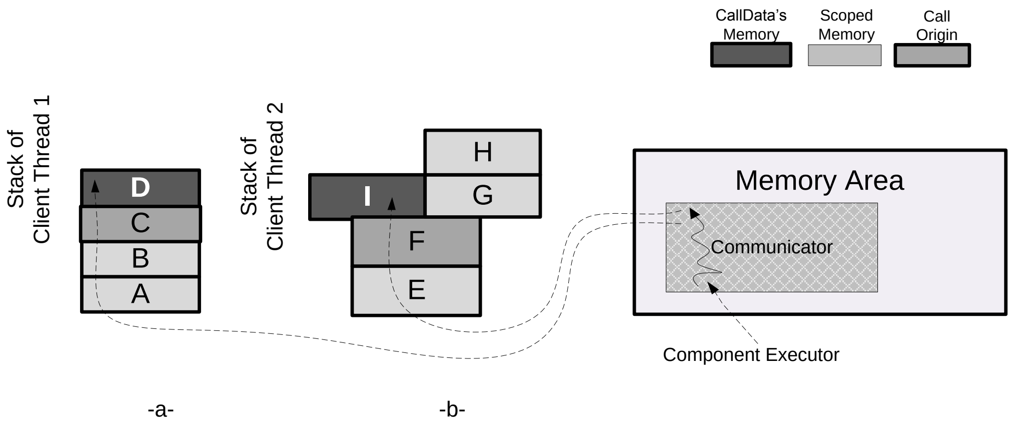


Figure 8-7. Nested Model

Level of abstraction: In both the above scenarios, each one of the the temporary scoped memories D, and I, is acting as an email box, which holds the shared call data between the caller thread and the call-executor thread. According to the RTSJ rules, these two scope stacks have to have at least one active thread running within them to keep them alive; otherwise, they will be reclaimed and data inside them will be deleted. So, in both scenarios, the calling thread has to keep the temporary scoped memory area within its scope stack during the call processing untill the result is delivered back to them. This behaviour is more suitable for the *synchronous* pattern of method calls; where by default, the calling thread blocks waiting for the result of the method call. However; for the *Future* pattern of method calls, the calling thread delivers the call to another thread to process, and it continues execution instead of blocking to receive the result. Then the call-processing thread can deliver the result to a certain future object that is accessible by the calling thread. In this manner, the calling thread can continue execution, and later, it can get/check the result from this future object.

Supporting this behaviour in the above two scenarios requires careful consideration and restriction, where, the calling thread can use either the `executeInArea()` method to execute within one of the inner memory areas in its memory scope stack, or it can enter another nested memory area above the temporary scope. In the first scenario the calling thread cannot leave the scoped memory D before getting the result of the remote method call, but it can use the `executeInArea()` method to run in either A, B, or C scoped memory areas. According to this programming model, the calling thread becomes tightly coupled with the temporary scoped memory area, which adds more complexity to the programming model, and makes it difficult for the programmer to organize his code. Also, in the second scenario above, if the calling thread, after initiating the Future remote method call *RM*, it wants to move to the H scoped memory area to execute some local method *LM* there, then, it has to call `F.executeInArea()` that executes a runnable, which has to call the method `F.enter()`, which itself has another runnable that has to execute `H.enter()`. If the logic of the method *LM* wants to check the result of the method call *RM*, then, the calling thread has to re-enter the memory area I again by using the sequence of calls `F.executeInArea()`, which, in turn, executes a logic that include the call `I.enter()`, then, the calling thread checks the result in the *I* memory area. If the future object has to be copied to the H memory area, things becomes more complicated. Hence, although using this model can be applied, it is more suitable for synchronous calls, while for future calls, although it can be used, it adds a lot of complications to the programmatic model, and makes it difficult to calculate the worst case execution time of the calls; in addfition to that, it complicates the analysis of the system.

Restrictions: In our proposed stub model, there is a critical restriction on using this model, this restriction is that the use of a queue that holds the call's temporary memory areas, D, and I, in the above scenarios, means that the references of D and I have to be accessible from the memory area in which the queue object is allocated, where this memory area has to be shared among all the threads' stacks, e.g. the container memory area. This means that the objects of D and I have to be created from the queue's memory area as well, or a lower scoped memory area in its scope, and they cannot be created from within any inner scoped memory area.

The support of the model in the forked memory model: In the above scenario, as we assume that the executors are configured to be initiated in the container's memory area, ContMA; then, in order that the executor be able to access the call's data, it has to enter the D memory area, and according to the RTSJ's single parent memory rule, the only way to do this it to build a scope stack that is identical to the portion of the calling thread's stack, from the outmost scoped memory area up to the temporary memory area, and attach this scoped memory stack to its own stack, i.e. the executor has to enter the memory areas A => B => C => D for the first scenario and E=>F=>I for the second scenario in sequence in order to process the call. This means that there is a need for a sequence of a variable number of nested operations to make this execution; this is supported in the forked memory model by the runnable stack pattern. The restriction on allocating the temporary memory within the ready queue of the stub limits the use of the reusable stack pattern, which assumes that the dynamically created temporary memory areas are created within their parent scoped memory areas.

B- Nesting within the Outmost Scoped Memory Area of the Calling Thread's Stack

In the previous model, the level of abstraction of the programming model is dependent on, and proportional to, the structure of the calling thread scoped memory stack. This is because the temporary scoped memory area can be nested within any memory area in the scoped memory stack, so the executor has to enter all the scoped memory areas below it in order to enter it. To overcome this problem, the temporary scoped memory area has to be created within a fixed place in the scoped memory area stack, which can be easily accessed from anywhere in the stack.

Scenario: In this model, we assume that the creation of the temporary scoped memory area is always to be nested within the outermost memory area of the scoped stack of the calling thread.

Level of abstraction: The choice of the outermost scoped memory area as the parent has better level of abstraction because:

1- The calling real-time thread has at least one memory area and at most one outermost memory area.

2- The creation of the temporary scoped memory area on top of the outermost memory area in the stack is done in one step exactly from within the current memory area, by creating it from within a runnable logic that is executed by calling `outermostMA.executeInArea()`

3- Entering the temporary memory area from the current memory area of the calling thread is done, at the most, in two operation in sequence; first by calling `outermostMA.executeInArea()`, in case the calling thread is currently running in a memory area in the stack other than the outermost memory area, then followed by a call to `tempMA.enter()`.

For example, in Figure 8-8-a, the calling thread, which is running in the C memory area, enters the temporary memory area (I) by calling `A.executeInArea()`, which executes the method `I.enter()` in its runnable logic, to enter the temporary memory area.

Similarly, in Figure 8-8-b, the calling thread running in E enters the temporary memory area H, by running `H.enter()` from the runnable logic which is first executed by `D.executeInArea()`.

4- Entering the temporary memory area by the executor; requires the executor to be able to do the two memory entries A=>I in the first case, or D=>H in the second case.

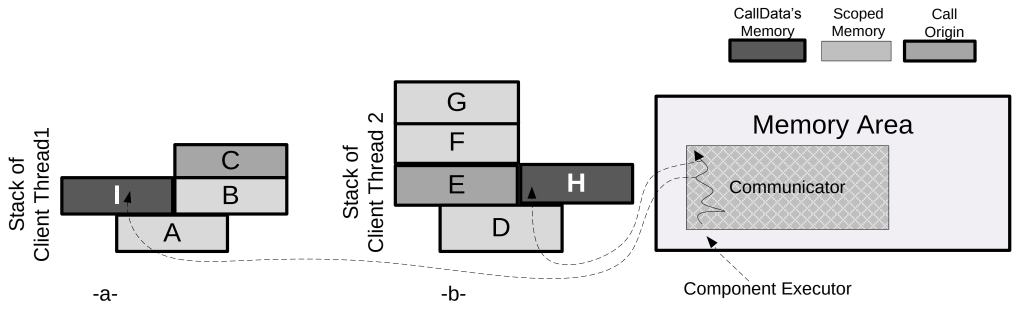


Figure 8-8 Fixed Nesting

As seen above, using this model makes the programming model more systematic and convenient, as it is independent of the structure of the calling thread scoped memory stack, because it fixes the scope stack depth below the temporary memory to be equal to only a single outer scoped memory area, where this single outer scoped memory area is fixed and does not change for the same calling thread.

Restrictions: Using this model for making future calls requires the calling thread to have a reference in the D/H memory area after putting the call data in it. This means that the calling thread has to run `executeInArea()` call from within the D/H memory area, in order to return back to the original memory stack, then, one or more `enter()` calls has to be mad to continue execution in the previous memory area, this adds a lot of difficulties on the developer to manage his code.

The support of the model in the forked memory model: The use of the container memory area as an initial memory area for both the calling thread and the executor, i.e. the outer most memory area, makes the length of any reusable stack pattern, which is used to access the temporary memory, is always of size two, e.g. ContMA=>TempMA, which gives better predictability in memory access time of *O(1)*. Also, the allocation of TempMA in this model has to be in the container memory area by default, which makes referencing of it from the ready queue in the stub, is allowed without any restriction.

C- Independent Temporary Scoped Memory Model(s)

Although the above model has overcome a lot of the difficulties that exist in the first model, it still has some problems regarding the sharing of the temporary memory with the executor and/or other threads. In addition to the difficulty in managing the code to continue execution in the previous memory area, another problem occurs within a scenario in which the calling thread can make a future remote call that will be executed remotely, and its return value will be saved in the temporary memory, in order to be accessed by another thread other than the calling object/component. In this case, the lifetime of the temporary scoped memory area will depend on the thread that is waiting the result, not the thread that initiated the call. In the above two models, the calling thread was working implicitly as a wedge thread for the temporary scoped memory area, but in this scenario, the lifetime of this thread can be shorter than the lifetime of the temporary scoped memory area. So, it becomes possible that the calling thread finishes its execution, and the temporary scoped memory area is claimed before the other thread that will access it to get the result enters it. In order to overcome this problem, the previous model of the temporary scoped memory area can be enhanced by providing a mechanism for controlling the life-time of the temporary scoped memory area explicitly, by one of the subcomponent(s) that use the lifetime control patterns discussed earlier in this dissertation in chapter 5, i.e. either by using wedge thread, fork thread, dual-fork thread, or memory pinner sub-component.

I- Using the Fork/dual-Forked pattern

The diagram in Figure 8-9, shows an example of this model as a modification of the previous model, by using the forked memory pattern, where in this model there are two thread scoped memory area stacks, where we assume that the first thread enters the forked scoped memory D, and the second thread enters the temporary scoped memory area G; these two memory areas are created on top of the current memory area C, F of the two threads respectively as follows:

Scenario: In this model, once the calling thread exits D/G after imitating the future call, these two memory areas should not be reclaimed, as the life times of these memory areas are assumed to be independent of this calling thread, as their life-times are controlled explicitly by the programmer, and any other thread including the communicator component's executor can access them, as long as the RTSJ memory access rules are satisfied.

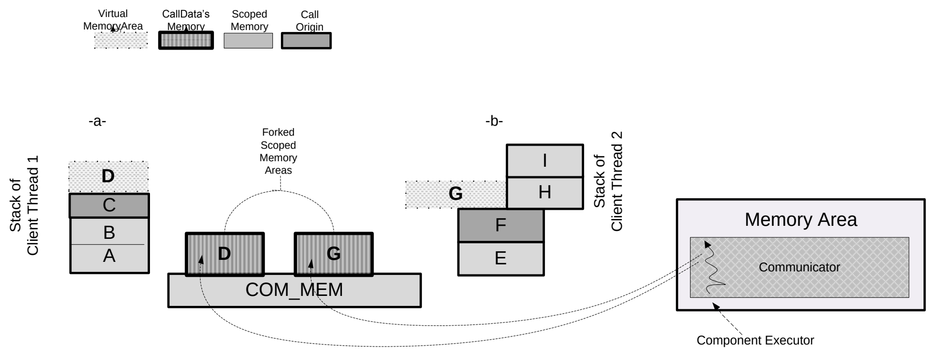


Figure 8-9 Independent Forked Temporary Scoped Memory Model

Level of abstraction and Restrictions: An important restriction of this model is coming up due to the need for COM_MEM to run this pattern. This is because if COM_MEM is chosen to be a scoped memory area, then, it has to be in the scoped memory stack for both the calling the thread and the executor thread and any thread that would access this memory area, and in the same time the section of the scope stack above it, up to the forked memory area, should be the same, in all the threads that needs to access this memory area, or the other option, is to let the COM_MEM to be one of the primordial memory areas, i.e. immortal/heap memory area. In addition to that, as the D, G can be of any depth within the scoped memory area stack, then the complexity of using the fork/dual-fork pattern will be *O(m)*, to hold a single temporary memory area, and increases to *O(nxm)*, in case of holding *n* memory areas. So, as it was the case in model B, to reduce the complexity and access time, we can assume the allocation of the temporary memory areas to be nested within the outermost memory area of all the threads concurrently accessing it, this will reduce the access time to be of *O(n)* for n different temporary memory areas.

The support of the model in the forked memory model: The forked memory model offers a good support for this model, as it assumes that all the schedulable objects are originated from the same memory area, the container memory area, ContMA. Also, the allocation of the temporary areas in the same container memory area ensures that that the depth of the forked memory to be one, which limit makes the access time to *O(1)*. In addition to that, the support of the forked-thread and dual-fork thread as sub-components within the container provides the pattern as a ready service to save the time of the implementation. Moreover, the fork/dual-fork subcomponent can itself be used as the ready queue that is used by the stub to save the calls that are to be executed by the call-executor provided by the communicator component.

II- Using Pinnable Memory

A similar but more simplified model of the above model can be built using the pinned memory areas instead of the forked model; see Figure 8-10, as there will be no need to have a COM_MEM as a common parent for all the temporary memory areas.

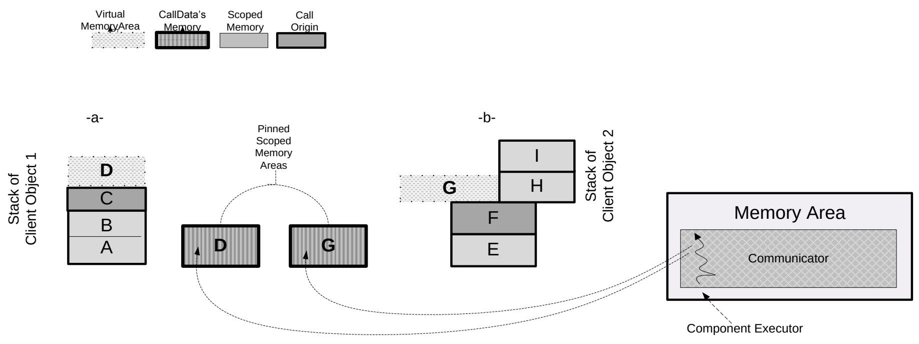


Figure 8-10 Independent Pinned Temporary Scoped Memory Model

Scenarios: In the first scenario, as shown in Figure 8-10, D and G are the temporary memory areas created for the future call, decrementing, they can be pinned once the calling thread leaves them.

Restrictions: The access to the pinned memory area in this scenario still have the restriction that the threads that access this memory area concurrently have to be subject to the memory access rules, i.e. they have the same scope stack, or at most one of them has a scope stack, while the others are running immortal/heap memory area, and they do not run in any scoped memory area. We can reduce these restrictions, by nesting the temporary scoped memory areas within the outermost memory area of all threads that concurrently access it. Another restriction is that this model is restricted to the RTSJ release 1.2 and not available in RTSJ 1.1.

The support of the model in the forked memory model: Our model provides a support similar to the support of the forked/dual-forked model discussed above, with an access time of *O(1)*, which is more predictable than the fork-thread, dual-fork model.

III- Using Wedged Memory

This model is identical to the pinnable memory pattern, but with the use of a new thread for each temporary scoped memory created inside it, which represents the major drawback of this pattern, because of the heavy cost of using the wedge threads on the system on the resources, scheduling and concurrency models, which is of order *O(n)*.

IV- Using Reusable Objects in Immortal Memory

In all of the above approaches, we assumed the use of scoped memory area as a temporary storage to store the remote call's data objects, so that it can be freed once the call execution is completely finished. Another approach that aims to use the Immortal memory area as the memory context in which these data are saved is a model that uses the reusable objects. The use of the reusable objects require that the data, which is required to be saved, has to have a common format that can be filled each time it is to be reused, i.e. the call data has to be represented as parameters that are saved within these objects, where these parameters are changed for each future call. So, the validity of building this model is dependent on the representation of the call's data within it.

A set of disadvantages that are facing this model, in addition to the restrictions defined on it in our reusable objects allocator sub-component, includes:

- **The high consumption the memory area resources.** To be able to process multiple concurrent future calls, there should be several reusable objects created and ready for use in the memory, where the number of these reusable objects is determined by the degree of concurrency required. As all these objects are not necessarily used the at the same time, as long as the degree of the concurrency is less than the maximum; hence, a considerable amount of the size of the allocation memory area will be reserved for use, but without actual need for it.
- **The internal fragmentation of objects.** If we assumed that the call parameters and any other remote call's data are serialized into reusable byte buffer objects in the immortal memory area, and they have to be ready for processing them by the communicator component's executor, then the sizes of these byte buffer objects have to be big enough to accept the worst size of the passed parameters' objects. However, these bytes objects are not always filled, so, each time a remote call is processed, the required size of the serialized parameters can be less than the assigned byte buffer's size. There causes a memory fragmentation equal to the remaining unused bytes within the byte buffer.

8.4 Empirical Evaluation

In this section, we present the results of some experiments that have been done using the new patterns and models presented in this thesis. In these experiments, we measure the performance and the predictability provided by these patterns and models and how using the RTSJ features have helped to improve these features.

All the experiments have been done using our modified implementation of the RMI-HRT open-source, which was built on a modified version of the jRate implementation. This implementation is running over the open-SUSE 11.2 Linux distribution, which runs Linux kernel version 2.6.18. The experiments have been done using a Dell laptop system, which has an Intel® Core™ 2 Duo Processor T6400 (2.0 GHz, 800 MHz FSB, 2 MB L2 Cache), with 4GB of 800MHz Dual Channel DDR2 SDRAM, and 500GB Hard disk.

8.4.1 Comparing DualFork and WedgeThread Patterns

In this thesis, in chapter 5, we presented a set of new patterns for managing the lifetimes of any defined set of scoped memory areas. Among those patterns, we considered that the DualFork pattern can be the best replacement of the commonly used wedge pattern, for the reasons presented in section 8.1, where this pattern itself is composed of other patterns presented in the same chapter e.g. the Reusable Objects Pool pattern. Hence, the main aim of this experiment was to compare the predictability and performance of these two patterns, when they are used to keep a set of scoped memory areas alive, considering that they are both using the RTSJ. In the same time, we need to compare the efficiency of the two patterns, when they run both on single processor and multi-processor systems, as both patterns are multithreaded patterns.

So, we ran this experiment twice; the first time was on the dual core system defined earlier; then the same experiment was repeated on the same system, but with setting the CPU affinity of the system to run the experiment on a single processor of the system, by using the Linux command *taskset*.

The idea of the experiment is to measure and compare the time required for both patterns to propagate into a set of scoped memory areas, where the experiment is repeated with different sizes of this scoped memory areas set.

The diagrams in Figure 8-11 and Figure 8-12 show the results of our experiment the time taken by both patterns to hold the required memory areas, where the horizontal axis represents the number of held scoped memory areas, while the vertical axis shows the time taken to hold these number of memory areas measured in micro seconds. In the case of the wedge thread pattern, as there are one wedge thread is used to hold each scoped memory area, the time is measured from the moment the first wedge thread is created, until the moment at which the final wedge thread stops waiting in the required scoped memory area. On contrary, in the case of the Dual-ForkThread, the time is measured between two moments; the first moment is when the Dual Fork thread is notified for the first time to update the held scoped memory areas list, i.e. the moment after adding the first scoped memory area, which causes one of the inner ForkThread starts to propagate, while the second moment is the moment when the Dual Fork Thread stops propagation and waits at the final scoped memory area.

The results in Figure 8-11 shows the measurements taken in case of the single processor system, in this figure it is clearly seen that the average time of using the Dual-Fork thread pattern is less than the average time of using the wedge thread, this is because the usage of the wedge thread pattern involves the creation of new threads, which is a heavy process and takes relatively longer time, while the use of the Dual Fork thread pattern assumes the usage of only two pre-created threads, which avoids completely the creation of any new thread. For the same reason, as shown in the diagram, the predictability of the required time for the Dual Fork pattern is much better than that of the wedge thread pattern.

The results in Figure 8-12 shows the results of the same experiment, but with the measurements taken when the experiment was run on the normal system, i.e. using two processors. Comparing the diagram in this case with the diagram of the single processor case, we find that the behavior of the two patterns is the same for the same reasons, but with a better predictability for the Dual Fork thread, as the usage of a single processor can affect the pattern, as there will be a need to do context switching from the currently running thread to one of the two inner threads of the Dual-fork thread, which may be not required in case of using multiple threads if there is a free processor to run the required thread.

8.4.2 Response Time of the Communicator Component

In chapter 6, we developed the design of our proposed real-time Communicator component, which is a component that can be used to support different levels of remote communication in real-time middleware. In the design of this component, we used the Components framework provided in chapter 5, which include a set of patterns that are built to achieve two main requirements;

- Providing a new and/or modified set of RTSJ-based design patterns to be used in building components compatible with the RTSJ memory model, and at the same time it benefits from using its efficient integrated scheduling memory to enhance the predictability.
- Hiding the difficulties of developing and using the RTSJ components, which arises when the new RTSJ memory model is used, as this memory model requires a certain set of rules to be respected to manage it, as discussed in 4.2.2 .

In addition to that, this component framework is integrated at the same time with the non-blocking strategy of non-blocking communication, as presented in 6.4, in order to provide a real-time behaviour, as well as to support several efficient communications mechanisms.

Hence, in this experiment, we aim to see how the usage of the RTSJ scheduling model, and the RTSJ-based patterns, which were assumed in the component framework, on the performance and predictability of the communicator component, when it is used for making calls, as it is the main element in our real-time middleware model, in both its sides; the client side and the server side. Also, at the same time, we need to compare the efficiency of the component when it is configured to use two different communication mechanisms, i.e. the non-blocking and the blocking modes, and see again how the new features of the RTSJ enhance the predictability of these two models.

So, we built a simple client-server test program that uses the communicator component at the both sides; the client side and the server side. This simple test program sends a packet from the client side to the server side, once the packet is received at the server side, it is sent back again to the client. This process was repeated many times, and the response time was measured each time, where the response time is measured as the time between two moments; the moment at which the program starts to send the packet from the client side to the sever, and the moment at which the same packet is received at the client side after receiving it back from the server.

The same experiment was repeated with two different configurations of the communicator component, the first configuration is the use of the *emulated blocking* mode, and the second configuration was using a *non-blocking* mode. Also, to see the effect and enhancement that result from using the RSJ scheduling model on the predictability of the response time, we have taken the response time measurements, for each of these configurations, with and without activating the RTSJ scheduler.

The results coming out of the *emulated blocking* and *non-blocking* configurations, in both cases of activating and not activating the RTSJ scheduler, were collected and drawn in the diagrams shown in Figure 8-13, and Figure 8-14 respectively, while in both these diagrams the vertical axis represents the measured response time, while the horizontal axis shows the iteration number of sending and receiving the packet.

In the blocking mode configuration, as seen in Figure 8-13, the jitter in the response time was bounded in both cases of using and not using the RTSJ scheduler. However, in case of using the RTSJ scheduler, the response time of the communicator component was bounded within a smaller range than the jitter measured in case of *not* activating the RTSJ scheduler. In addition to that, the average response time in case of using the RTSJ scheduler was much better, than the case of not activating the RTSJ scheduler, this is because activating the RTSJ scheduler ensures that the execution of all the test program threads will be scheduled in according to their real-time priority, which was set higher than any background task, this ensures in turn that the execution of the sending and replying the packet at the client and the server sides will not be preempted by any background task; hence, the response time will be faster. This means that, as assumed in our hypothesis, the patterns provided in the component framework can be integrated with the RTSJ new features such as the memory and scheduling models, to enable the creation of components with predictable behaviour, e.g. the bounded jitter in the response time as proposed here.

In the case of the non-blocking mode, see Figure 8-14, we can see the following:

- By comparing the average response times in both cases of activating and not activating the RTSJ scheduler, we find that in this mode, i.e. the non-blocking mode, the average response times are very close, while there was a clearer enhancement in the average response times in the case of the blocking mode. This is due to the efficiency provided by the non-blocking mode of communication, for the reasons discussed in section 6.3.2.
- As seen in the blocking mode, using the RTSJ scheduler provided a predictable execution and bounded jitter in the response time of the component that adopts the proactive model of execution, i.e. uses the non-blocking mode of communication, as it guarantees that the execution of the real-time threads constituting the component that execute the non-blocking methods to be according to their defined priority. This is a great enhancement of the usage of the non-blocking mode, which the lack of order of execution was one of its main disadvantages, as discussed in section 6.3.3.

8.4.3 The Priority Assignment of the of the Communicator

The thread that runs the polling loop within the Communicator component is the main thread of this component, and as this component is built using the RTSJ, then the assignment of a certain priority of this thread is an important issue as RTSJ scheduler uses this value to determine the order of the execution of this thread relative to the other three in the system; hence, the jitter of the response time can be affected by the priority value assigned to this thread. Hence, the aim of this experiment is to see the effect of assigning a priority value to the polling thread of the Communicator component on the jitter; and hence the predictability, of the response time of this component.

In order to see an effect of the assigned priority value, we built a simple test program that consists of a single client and a single server, and we measured in this experiment the response time of sending a packet from the client to the server, and then receiving it back, exactly in the same way used in the experiment presented in section 8.4.2. The response time measurements have been taken using two different values of the priority value of the polling thread; the first time the value was assigned to be greater than the value assigned to the client thread, while in the second time, the priority value was assigned a value smaller than the priority of the client thread. The choice of the client thread priority as a reference is because the client thread is the thread that initiates the calls over the Communicator component, and its scheduling for execution is affected directly by the execution of the polling thread in the Communicator component, especially if the program runs on a system with single processor, as both of these threads, with all other threads in the test program, are scheduled by the preemptive priority scheduler provided by the RTSJ.

As seen in the diagram in Figure 8-15, in the case of assigning a priority of the communicator component less than the priority of the client thread, the average and the jitter of the response time are lower than when this priority was assigned to be more than the client thread priority. This is because that with each iteration to send the packet to the server, the scheduler executes the real-time threads that runs in the program according to their priority, and as the client thread has higher priority than the polling thread of the communicator; then, in the single processor case, any currently running thread with lower priority has to be preempted, and as the polling thread has lower priority, then it has to be preempted, and this polling thread, which has the lower priority, would be interrupted by the client thread in each iteration, in order to

8.4.4 The Predictability of the Communicator in the Case of Multiple Clients

In the previous experiments, the test program was built as a single-client/single-server model, in this experiment; we need to see the effect of using multiple clients on the predictability of the component, by making two concurrent calls to the same server, by two different clients to the same server.

Hence, we used a model that uses our Communicator component, where the component was configured to run in the emulated blocking mode, the model consists of two instances of the client side of the same test program presented earlier, these two clients use periodic threads that send two packets simultaneously to the server side; one packet from each client. These packets are received and replied back to the two calling clients using a single server, where the release parameters of the real-time threads of the two clients were assigned to start simultaneously, and they are assigned the same period of execution. As this model is a multithreaded model, we tested this model on our two-processor system using two different configurations;

- No external setting of the CPU affinity is made, i.e. the clients and the server are scheduled by the system scheduler.
- The CPU affinity of the clients is assigned externally, where each client is assigned to one of the e processors using the taskset command.

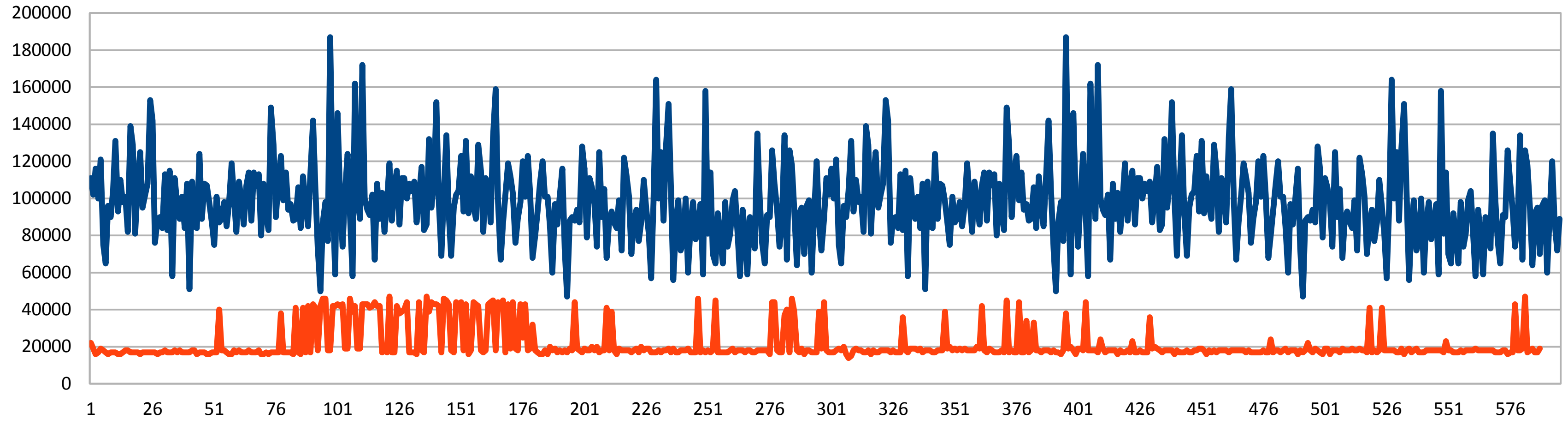
At the same time, we need to see the effect of using the RTSJ on the component predictability, on both of the above two configurations, and we need to see how the RTSJ enhances the predictability of the Communicator component and at the same time it guarantees the clients to execute according to their priorities, i.e. clients with higher priority execute before clients with lower priority. Hence, the experiment was tested for each configuration two times; the first time while the RTSJ scheduler was activated at all the system instances, i.e. the client sides and the server side instances, and the second time while the RTSJ scheduler is not activated.

The diagram shown in Figure 8-16 shows the results obtained by running the test program with the CPU affinity of each client thread is set to use a different processor of the system, and without the activation of the RTSJ scheduler. It is clear in this figure that there are relatively large boundaries of the jitter in the response times of both threads, whereas the average times of the response time of the two clients are close to each other. This is expected with these configurations as the missing of a real-time scheduler makes the two clients preemptible by the Linux scheduler by any other thread running in the system without respect to their higher real-time priority.

To see the effect of using the RTSJ on the component model, the RTSJ scheduler was activated and the results were obtained and drawn in the diagram in Figure 8-17. In this diagram, we see that the values of average and jitter in the response time were reduced significantly for both threads than the values shown in shown in Figure 8-16, which were obtained when the RTSJ was not activated. This shows clearly that using the RTSJ scheduler enhanced the predictability, even with multiple concurrent executing clients. In addition to that, we note that the jitter and average time of the client thread with higher priority is much lower than the corresponding measured values of the client thread with lower priorities. This means that the priority of the caller is respected when the calls are made in our model.

The above two tests were repeated without setting the CPU affinity, i.e. the system scheduler can allocate the client thread to any available processor. The diagrams in Figure 8-18, Figure 8-19 show the measured response times of running these two tests. It is clear from the figures that the observations are similar to the observations that were obtained when using the CPU affinity still exist, i.e. activating the RTSJ scheduler enhances the predictability of the model.

In addition to the above, in Figure 8-19, when no CPU affinity setting was set and the RTSJ scheduler was active, we observe that the jitter in the response times was nearly bounded within the same boundaries for both clients threads, which was a result of using the RTSJ scheduler as explained earlier; in the same time, the average response time for both threads were nearly equal, although one of the two clients has a priority greater than the second one, this equality is because the configuration in this case enables the system scheduler to use all the available processors in the system; hence, the thread with lower priority needs not to wait for a thread with a higher priority as long as there is available processor in the system to which it can be allocated. However, when both clients' threads were restricted to use the same processor in the second configuration, as seen in Figure 8-17, we found that the thread with higher priority has a lower average response time than the thread with the lower priority; also, the jitter in the response time was much less, which means that the communicator component respect the priorities of the clients' threads; hence, using the Communicator component in our middleware guarantees that the clients calls are guaranteed to be executed according to the priorities of the calling clients.



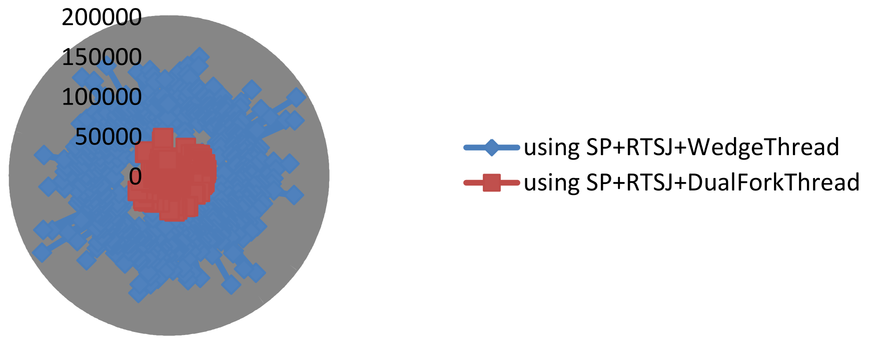
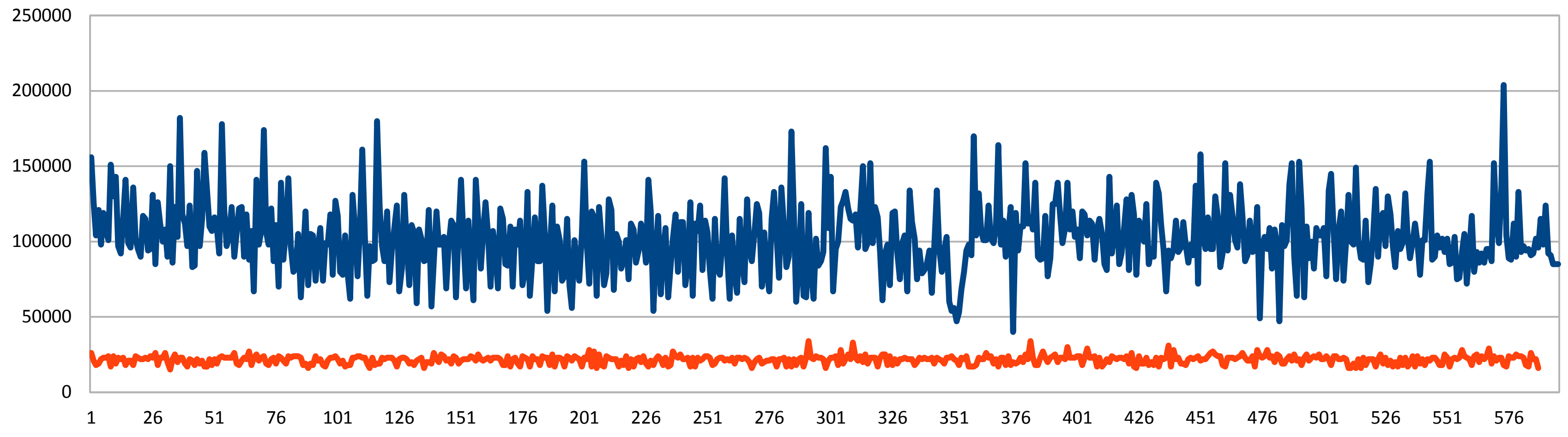


Figure 8-11 Comparing Dual Fork and Wedge Thread updating Times [SP]



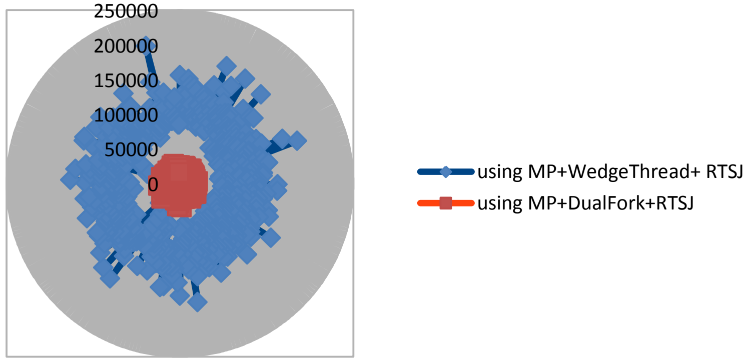
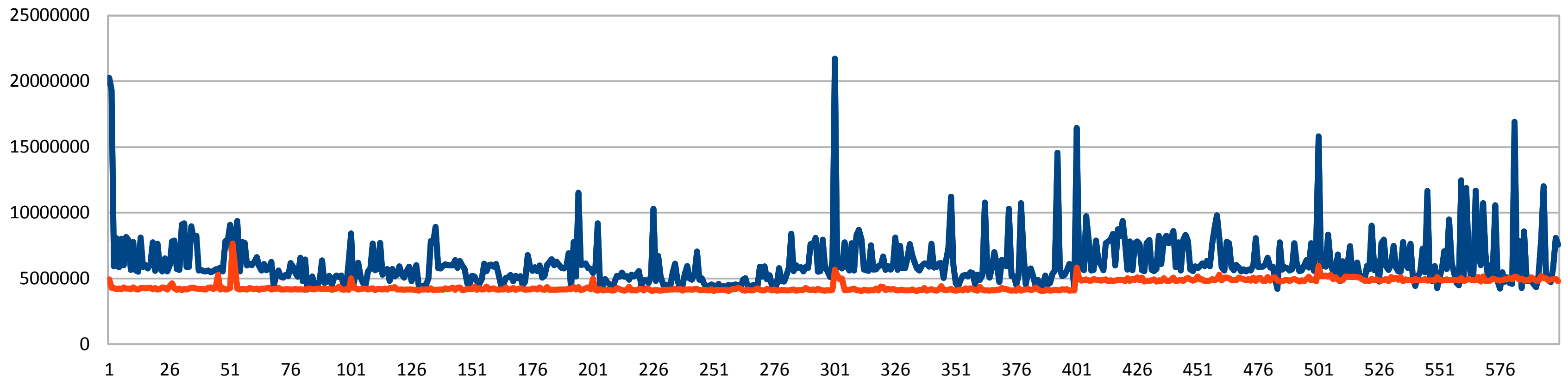


Figure 8-12 Comparing the updating Times Predictability of both the Dual Fork and Wedge Thread [MP].



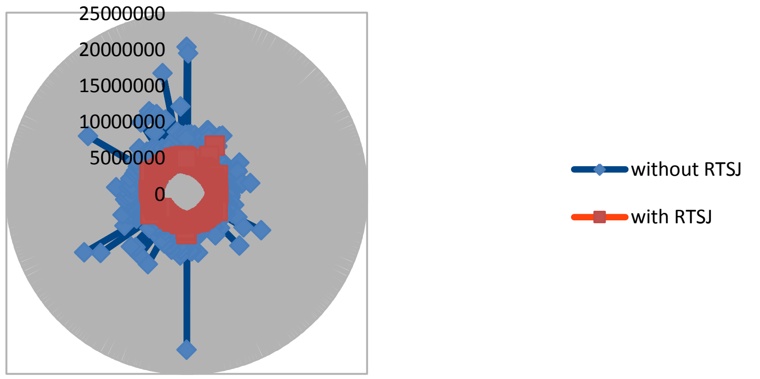
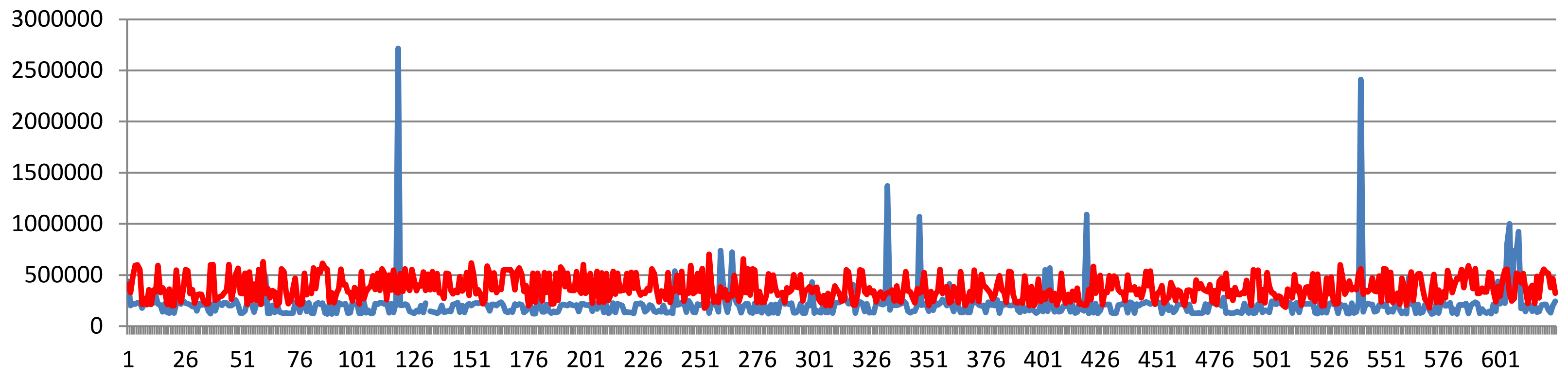


Figure 8-13 Emulated Predictability of the Response Time [Blocking Mode]



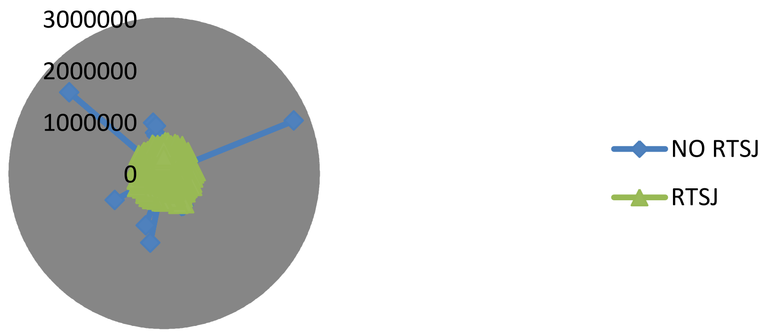
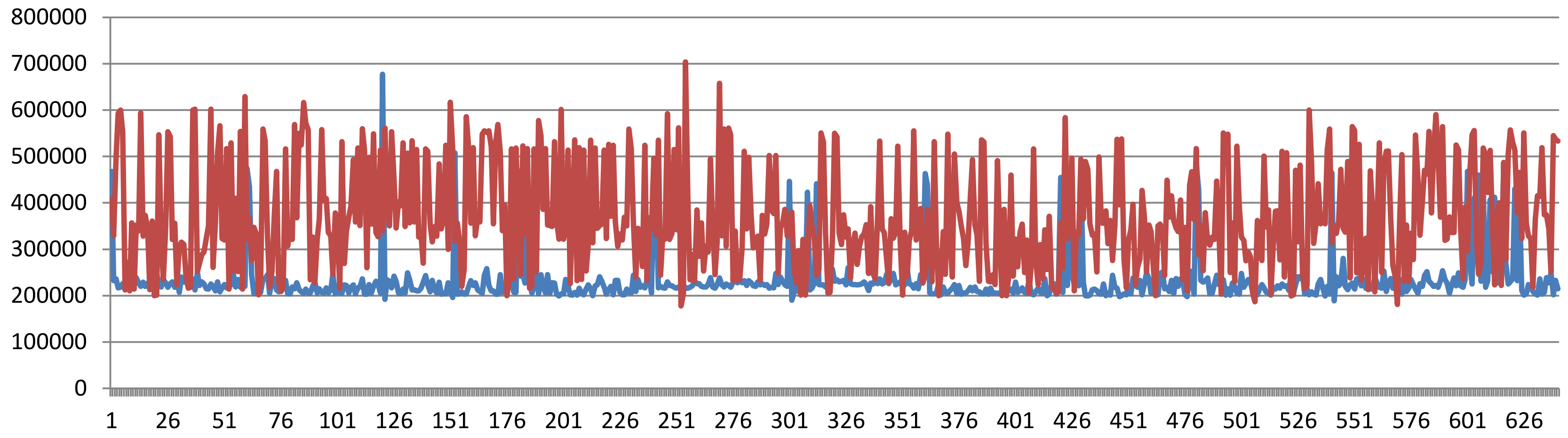


Figure 8-14 Emulated Predictability of the Response Time [Non-Blocking Mode].



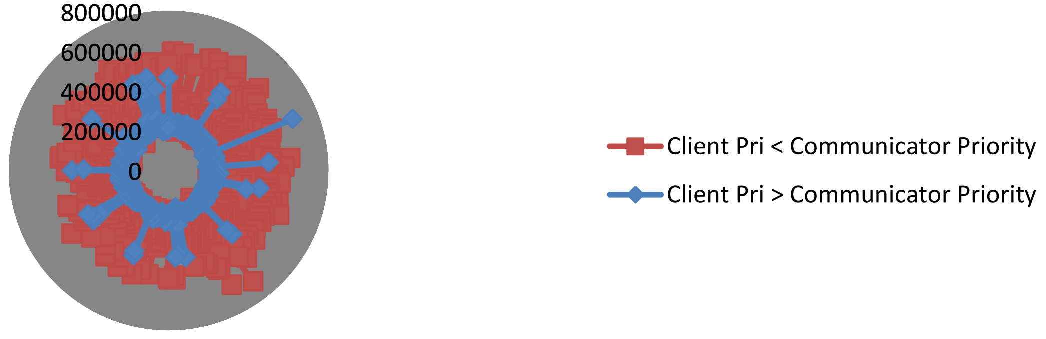
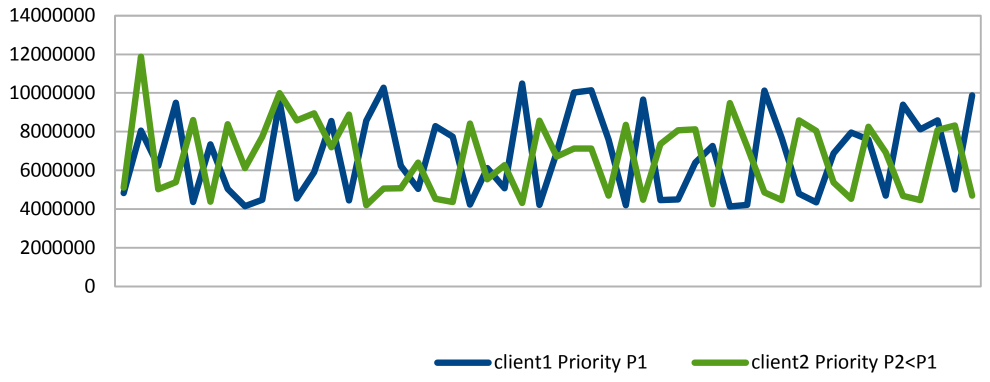


Figure 8-15 Predictability of the Response Time [Non-Blocking Model].



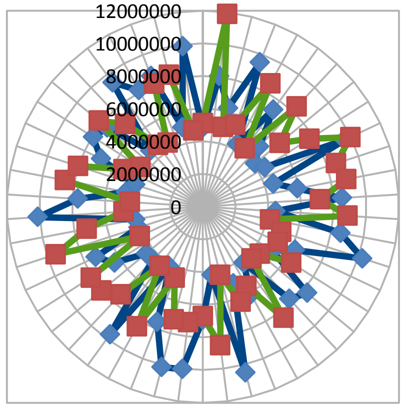
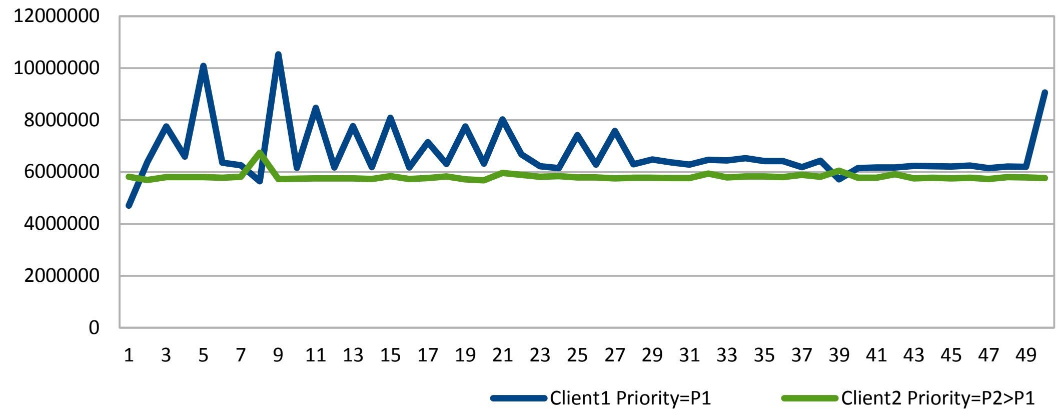


Figure 8-16 Response time [NO RTSJ+SP+Blocking Mode] of two concurrent executing clients where client1 priority (P1)> client2 priority (P2).



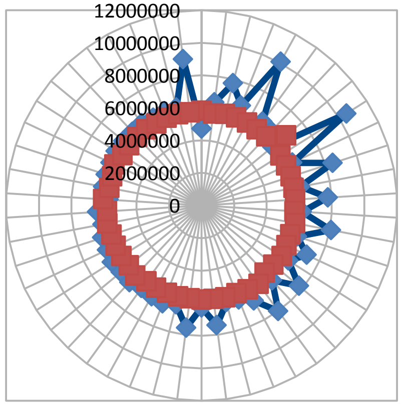
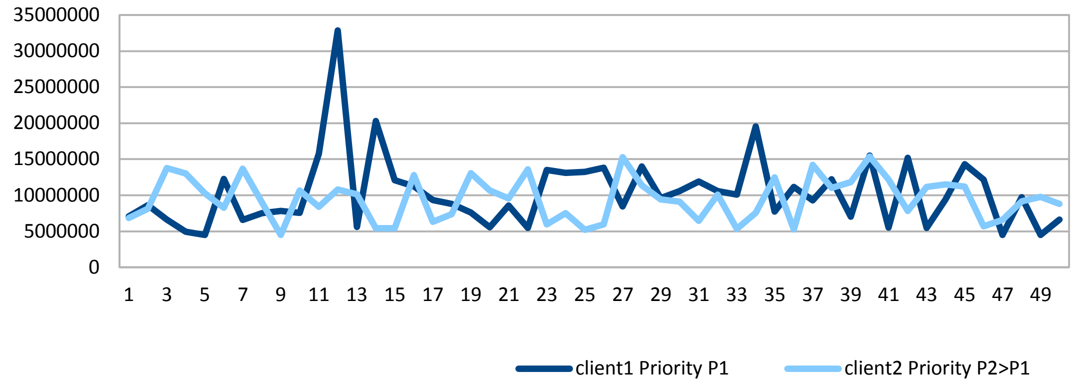


Figure 8-17 Response time [RTSJ+SP+Blocking Mode] of two concurrent executing clients where client1 priority (P1)< client2 priority (P2).



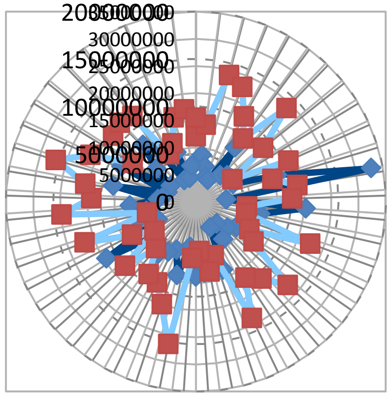
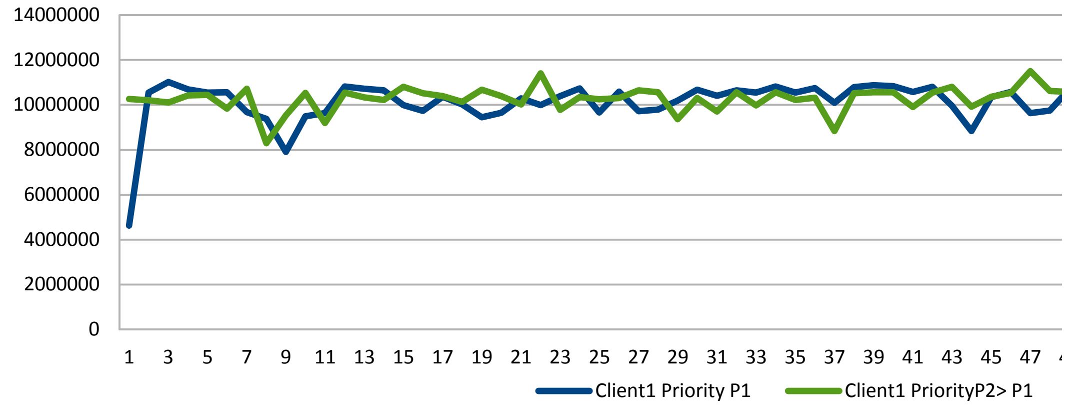


Figure 8-18 Response time [No RTSJ+MP+Blocking Mode] of two concurrent executing clients where client1 priority (P1)< client2 priority (P2)



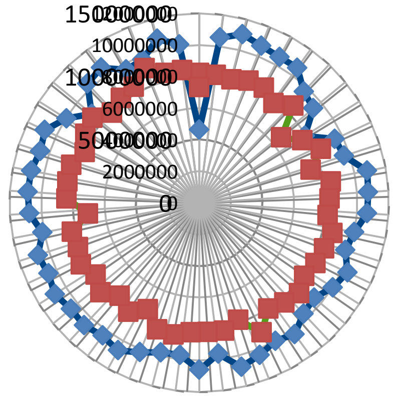


Figure 8-19 Response time [RTSJ+MP+Blocking Mode] of two concurrent executing clients where client1 priority (P1)< client2 priority (P2)

8.5 Summary

In this chapter, we evaluated the design patterns used in building the component model. We provided two kinds of implementation; (1) analytical evaluation of the pattern including their structures, timeliness, memory footprints, levels of abstraction and the generality of use, (2) Experimental evaluation of the main patterns and components of the real-time middleware, by measuring some individual metric values, e.g. response time(s) of requests, where a comparison of the predictability of these measurements which have been made on single processor and multiprocessor, and when RTSJ is used and when not used. Then, we studied the effect of using the Java NIO implementation in the JRate/gcj on the memory predictability of the system, and provided our solutions that have overcome these problems. Also, we evaluated the memory models for supporting the stub internal architecture, for making the remote calls.

Chapter 9

Conclusions

Distributed real-time systems are nowadays playing an important role in many industrial and commercial fields, and are expected to play a greater role in the future. In addition to the complexity of their structures, due to their inherited distribution nature, developing such systems faces many challenges to support their real-time requirements. Research in this field is widely increasing in many directions including; developing reliable communication networks, building predictable communication protocols, enhancing resource scheduling algorithms, and developing efficient programming languages and architectures that support mechanisms for building these systems, etc.

Most of the current middleware technologies are implemented with no consideration to any real-time requirements, as most of them are directed to non-real-time business environments. Hence, using the middleware technologies, in their current structures is not suitable for developing distributed real-time systems, which is required to be predictable and has to be able to satisfy the system's end-to-end timeliness.

One of the most active fields of research in this area is providing real-time middleware solutions that help build distributed real-time systems. Middleware has proved to be an efficient solution for building traditional distributed system in general, as it abstracts many of the underneath details of these systems, and hides their complexity; which in turn, speeds up the development process and provides more reliable and efficient systems. Many of such middleware solutions are implemented in Java, which has been proved to be one of the most efficient programming languages for building the distributed systems, due to its modern architecture that adopts the object oriented mechanisms to provide networking, communication, and distribution facilities with high levels of abstraction, which hides the complexity of building the distributed systems.

Middleware solutions that adopt in their structure the distributed object paradigm are widely used for developing distributed systems. CORBA and Java's RMI are the most common implementation of such paradigm. Most of the research toward developing real-time middleware targeted the language independent OMG's CORBA, as one of the main middleware solutions available. This research has led to the Real-Time CORBA specification for both static and dynamic systems. However, neither the Java RMI, nor the implementation of RT-CORBA in the original Java language, does offer the required predictability that enables using them in developing distributed real-time systems.

Other Java middleware solutions that use distribution paradigm, other than the distributed object paradigm, are available. Some of these middleware targeted real-time systems, such as JXTA and DDS. However, all current Java implementation of those middleware are based on the original Java. Hence, these middleware are expected to be subject to the inherent problems of the Java language. Implementation of such middleware solutions in RTSJ is supposed to be more predictable and reliable for real-time systems.

A recent direction of research in the field of real-time middleware is toward developing real time implementations, based on the RTSJ, that support building distributed real-time systems. This direction of research is based on the success of the release of the newly emerging real-time Java specification, the RTSJ. The RTSJ is developed to solve many problems inherited in the Java language that made it un-predictable and unreliable to be used in real time systems. Many challenges are facing the researchers to push the RTSJ to be used for the distributed systems, as it requires changes to both the underlying structure of the current Java distribution technologies, i.e. RMI, and to the Java virtual machine itself. Some research has already been done in this direction. Most of this research targets the RMI architecture, trying to analyse it to detect the non-real-time features of it and hence, modifying the structure to support the real-time features, but it is still far away from producing a full specification.

There is a trend in Software Engineering and industry toward building complex software systems by using reusable software technologies, such as components and design patterns, in order to ease and enhance the development process of these systems. So, many of the current middleware solutions adopt these technologies in their internal designs. However, applying these techniques to real-time systems, particularly the distributed real-time systems, is not a straightforward process, as the restricted timing and predictability constraints that are required for building real-time systems make it difficult or even impossible to reuse the existing conventional components and design patterns for building real-time systems. Hence, it is currently a big challenge, which faces the researchers and the developers of the real-time systems, to develop real-time reusable real-time components and design patterns.

The Java language has been used in developing many component technologies, e.g. Java Beans. However, due to the unpredictability of Java, the Java components cannot be used in building real-time systems.

The RTSJ presents to the Java language a set of features, e.g. new memory and scheduling models, to ensure the predictability of the execution, which is essential for building real-time systems. However, converting existing Java components and middleware to use the RTSJ is not straightforward. As the RTSJ models have a set of constraints, e.g. the single parent rule, that have to be considered and managed in order to be able to use it. Hence, this results in a need for presenting new models for components, where these component models should use a set of novel design patterns that act as controlling and managing patterns, e.g. Fork pattern, that hide the complexity of the RTSJ memory model, as well as they provide built-in real-time structural sub-components which provide real-time architectures for the real-time components including a reusable objects allocators, and reusable schedulable objects allocators.

It is also necessary to limit the blocking time of the low level communication layer used in building the real-time middleware communication solutions. This is a basic requirement for real-time systems to limit the priority inversion. So, adopting the non-blocking mechanisms within the communication layers is important to enhance the predictability.

There is a wide range of distributed real-time systems with different requirements and different levels of criticalness and timing requirements; hence, a single model of middleware is not enough to satisfy the requirements of all these systems. So, it is important to add high levels of flexibility and reconfigurablity within the real-time middleware solutions to ease using it in multiple distributed system types.

In order to conclude this thesis, we summarize in the following sections the key contributions of this research. Then, we identify a number of limitations of this work. After that, we discuss the future research directions in relation to using the design patterns and components in real-time middleware, as well as the support of more flexibility and reliability in their designs. Finally, we provide a brief note that concludes the key message that this thesis tries to convey.

9.1 Summary of the Key Thesis Contributions

The basic motivation for the work in this thesis is that the conventional middleware solutions are not useful for building distributed real-time systems, as the components and/or the design patterns used to build them are not supporting the required predictability, and they do not guarantee the end-to-end timeliness. A set of design patterns and components that offer these features can replace the existing ones in the current middleware solutions, to make them suitable for use in the distributed real-time systems. The RTSJ offers a set of new features that enhance building such real-time design patterns and components. So using the RTSJ to build design patterns and components for the existing Java middleware solutions, e.g. Java RMI, can make these Java middleware predictable and usable for building distributed real-time systems.

A key contribution for support building framework component-based real-time systems based on the RTSJ in this thesis is the provision of a component framework. In this framework, we proposed that a new memory component model and provided with it a set of sub-components for memory management, memory allocation and reusability, as well as sub-components for reusable schedulable objects, which are saved in thread pools within the component hierarchy and integrates with its memory model, and in the same time, it supports using the logic as a reusable sub-component to enable a high degree of flexibility.

For the real-time Java application developers, this thesis provides the *Forked Memory Model* as a key contribution for support building software components. The *Forked Memory Model* is a configurable model that is compatible with the RTSJ, and can be integrated with some of the following patterns, to build RTSJ-based sub-components/services for memory management services:

1- **The Multi Named-Objects Portals:** This is a pattern that enables the sharing of multiple objects among several threads running in the same scoped memory area through the portal object of this scoped memory area.
2- **The Fork Thread, the Dual Fork Thread, the Pinnable Memory:** These are a set of memory management patterns that can be used for managing and controlling the lifetime of the internal scoped memory area stacks of the forked memory model.
3- **The Executable Runnable Stack:** This is a pattern that can be used for the internal design of a task that runs within a single stack scope in the forked memory model.
4- **Reusable Objects Allocator:** To manage the allocation/recycling of objects **within** the component memory model.

Another major contribution of this thesis is the model of the configurable real-time communication component, which is presented in chapter 6. This component represents a low level networking communication layer that adopts the non-blocking mechanisms of communication as its basic model of networking communication. The non-blocking support ensures minimization of blocking time to avoid the priority inversion. The component can be reconfigured to support other forms of communications as well, to ensure the generality of use. The building of the component using the RTSJ ensures the avoidance of the unpredictability inherited in the Java language, due to the use of unpredictable garbage collector, where the internal design uses scoped memory areas and pools of real-time threads, or asynchronous event handlers, to handle the communication events occurring on the channels of the component. In addition to that, the component supports two modes, client mode and server mode, to ensure the flexibility and wide use of the component. The component itself is used as a sub-component within the container of the proposed component model, to support different forms of low level communication services to the components.

Finally, the last major contribution of this thesis is the presentation of our model of the real-time remote middleware in chapter 7, this model joined together the design patterns, the component models, and the communicator component presented in the earlier chapters in addition to some other design patterns, to build a new model for remote communication that offers high-level communications services within the component model with the following features:

1- **Reconfigurable models of invocations at the client side:** Where it supports the Poll Object pattern as a basic pattern for the invocation, and this pattern can emulate other invocation patterns.

2- **Reconfigurable models of call handling at the server side:** Using the communicator component enables the server object/component to handle the call, using the different models of communications supported by the component.

3- **Shared communication component layer:** Isolating the communication layer as a component ensures the ability to share this component among several clients**.**

4- **The support of different models of call execution parameters:** The model shows how the RMI protocol can use different models; i.e. client propagated, server centric.

5- **The predictable serialization:** The middleware adopted a predictable serialization protocol to ensure the predictability of the call transfer.

6- **Different models of stub execution:** We showed how the different lifetime management patterns of scoped memory areas, which can be integrated with the proposed remote middleware to make the calls, and we presented the different constraints of using them.

7- **Support for the Future Call invocation pattern:** We showed how the stub of the RMI model can use the services provided in the client component including it, to enable making future calls in the RMI model.

A final contribution was the presentation of a solution of the unbounded memory consumption in the communicator component due to the use of the Java NIO’s selector and selection keys objects, using the patterns and sub-component services provided in our component model.

9.2 Limitations of this work

The design patterns, the communication component, the real-time remote communication middleware are all developed in prototype versions in order to prove the hypothesis. Therefore, despite the contributions described above, there structures require extensive research in order to optimize their behaviour for reusability and building distributed real-time systems. Consequently, the RTSJ’s No Heap schedulable objects supported in the RTSJ requires a special consideration to ensure the avoidance of the using of the Heap memory. In our designs we have not deeply considered providing the exceptions and checking mechanisms required for handling these restrictions.

One major limitation of use in this work is the assumption of using the TCP/IP as the underlying networking protocol. This protocol offers a lot of unpredictability and can be used for many of the distributed real-time systems, especially the ones with high constraints on the networking communications. But, this does not mean the invalidity of using our work on the real-time networking protocols, if the required communication protocols are implemented on them.

Our model for the real-time remote communication is based on the RMI-HRT that uses static addressing of the remote server objects. This is important for a certain set of real-time systems that have a limited number of remote server objects. However, some distributed systems with real-time systems constraints can involve a dynamic number of remote servers; hence, there is a need for doing research on the mechanisms required to enable the use of the real-time dynamic locating and naming strategies for the remote servers. Initially this needed real time support can be provided by implementing these mechanisms in remote servers, which are implemented using the real-time middleware presented here.

9.3 Future Work

There are a set of directions for the future work uncovered in this thesis. The first of these directions is the provision of pattern language, which has to define design patterns compatible with the RTSJ memory and scheduling models, for supporting the build of real-time Java middleware. This is for two reasons; the first is that we have provided only a small number of design patterns that can prove the thesis hypothesis, but there are many patterns that are applied in developing conventional middleware solutions that need to be reconstructed, to be used in distributed real-time Java applications. The second reason is that there are other new features provided in the real-time Java, e.g. the Asynchronously Interruptible Events, requires special consideration to handle them, and it is important to see how the design patterns can support these new features.

The second direction is to complete the framework for the real-time Java components; where, in addition to enhancing the internal model presented in this thesis; this framework has to consider other new features of the RTSJ e.g. multiprocessor support. Also, the proposed components should have the ability to offer different levels of predictability and real-time support through their standardized external interfaces, by using contract-based techniques. Also, the management of the lifecycle of the components and the support of fault tolerance and dynamicity in them are to be considered, as well as extending the component model to support the mobility over the network.

The third direction of our future work is toward enhancing our model for the real-time Java RMI middleware. This includes the following:

1- The support for mechanisms that enable the safe use of No Heap schedulable objects.
2- Providing implementation of the model over real-time networking protocols.
3- Study in depth the distributed garbage collection in order to evaluate its predictability and see how it can be enhanced.
4- In addition to the support of the *Poll Object* invocation pattern in our real-time remote communication model presented in this thesis, we aim to study how an RTSJ compatible *Call-Back* invocation pattern can be supported as well.

As we focused in this thesis only on the Level-1 of integrating the Java RMI and the RTSJ real-time RMI middleware; hence, our fourth direction of the future work is toward extending the work to:

1- Support the Level-2 of the integration; by modifying the proposed architecture to enable the inclusion of the distributed thread model within it.
2- Supporting new architectures; there are many middleware solutions that are built over the RMI; e.g. Java's Jini, hence, we can investigate the architecture of these middleware to see what additional support that can be added to our model to build these middleware solutions. For example, in (Alrahmawy and Wellings 2007), we presented an initial middleware model for mobility based on the RTSJ. We provided in this model a set of algorithms and models for scheduling and migration, so we can integrate these models with the model presented in this thesis to build this real-time middleware.
3- Multiprocessor support; there is a trend in the moment for building RTSJ applications on multiprocessor systems, a possible extension of our work is to investigate how our model can integrate with the multiprocessor architecture, both for supporting internal communication among the processors that have no shared memory, and for the possibility of extending the design patterns used in this thesis, to be mapped to the memory architecture and the multi-processors of these systems.

Due to the diverse nature of the real-time systems types, including the distributed systems, it is not possible to have a single specification that covers all of these systems. There is a trend among the researchers working in the RTSJ to develop profiles of the RTSJ specifications for individual types of the real-time systems; e.g. Safety Critical Java. Hence, one of possible directions of the research is to investigate the compatibility of our real-time middleware model presented here to these profiles, to see how we can define individual specifications of our model for each one of these profiles.

9.4 Final Word

The distributed real-time systems are widely used in many industries. The technologies for building these systems should have high levels of predictability, and have to guarantee the end-to-end timeliness. Conventional middleware are basic for building distributed systems, but they cannot be used to build the distributed real-time systems.

The real-time Java community still has a lot of work to do toward developing full specification for distributed real time systems in Java, but we believe that building middleware solutions by integrating the RTSJ with the Java RMI is the basic step toward building distributed real-time systems in Java.

In the thesis hypothesis, we proposed that using the new features added to the Java language by the RTSJ with efficient networking communication mechanisms can help support building new design patterns and components, or changing existing ones, which can be integrated together for developing real-time versions of the existing middleware solutions.

From our work presented in this thesis, we found that it was very difficult and inefficient to directly use the RTSJ rules for building a real-time middleware solution. This is because the internal structure of such middleware solutions consists of many parts, and a lot of integration is required among these different parts. Adding to that, the restrictions of the RTSJ scheduling and memory access rules, makes it very difficult for the developer to manage the code and to gain the required predictability, especially that it is difficult for him to use the general design pattern to build these solutions, as these patterns do not respect the memory and scheduling models of the RTSJ.

On the other hand, when developing the component framework and using it for building the middleware parts made the development much easier, as building a set of patterns and component that are commonly needed in middleware solutions, and at the same time respect the RTSJ memory access rules in building them, makes it easier for any developer to use these patterns. In addition to that, using patterns such as the Forked memory model, and the Reusable Runnable Stack class, provides a simpler method for the developer to run his code in a stack of scoped memory areas, by hiding many of the memory access details, this in turn can help in reducing the code size and makes it easier to read and maintain. Also, the development of RTSJ specific patterns such as the Dual-Fork pattern helps to overcome the problems of managing the lifetime of the scoped memory areas in many RTSJ applications, not just the middleware. In addition to that, the development of the Communicator component as a general RTSJ component that supports the low level communication helps the developers to develop various real-time middleware solutions with different communication mechanisms, as shown in this thesis in the different design models presented for the RMI middleware.

So, the development of RTSJ based component framework, catalog of patterns, and component libraries, is a very important step for pushing the RTSJ to produce efficient and predictable real-time distributed and middleware solutions.

One of the main difficulties which faced this research is the missing of a reliable open-source implementation of the RTSJ. The RTSJ is relatively new and there are only few open-source implementations available. OVM(Grothoff) and JRate are examples of those open source VM implementations. These VM implementations are incomplete, i.e. they do not support all the new features of the RTSJ, or even they have incorrect implementation of those features. For example, we found that the early versions of JRate, which was used for the experiments presented in this thesis, do not implement the memory assignment rules correctly, which affected the initial implementations of the algorithms and patterns and made them have some bugs. Also, we found that the OVM does not support the JNI.

One more difficulty is that the implementations of the jRate and OVM are made on specific Linux distributions, and unfortunately, the implementations of some RTSJ features, e.g. periodic/aperiodic threads in jRate, are using some functions of these distributions, which were found to be incorrectly implemented and they have bugs and they are corrected in later versions of these distributions. Hence, to run jRate for example on other distribution, or even on a later version of the same distribution, we had either to reimplement those features using other methods, or avoid using them completely. Another major difficulty in our research was the migration of the RMI-HRT to the jRate, as RMI-HRT was built using an evaluation version of the Jamaica VM, this required us to make a lot of changes to the internal classes of the GCJ which is the backend of the jRate, to implement many of the missing features, particularly for the Java NIO packages. This has taken a very long time of our research, as we had to make a lot of tests and debugging just to find out the required missing features, and even we had to change among different jRate implementations, one of these main problems was in a jRate VM implementation, which uses MARTE-OS library, as we found that the threading model of this VM is a single threaded model, i.e. all the threads running in the VM are mapped to a single thread in the operating system, while our research is mostly based on using multithreaded model, e.g. a single blocking call to monitor the events result in hanging all the program, we spent a lot of time trying to avoid this problem by using external kernel threads for monitoring the events instead of using VM threads, but in the end we found that this is a nonreliable solution, and we decided to move our work to another JRate implementation, which is guaranteed to support the multi-threaded VM model, which was not an easy decision after spending many months developing and testing code on the single-threaded JRate VM.

From the above, we can see that there were many challenges and difficulties that faced us during the research that led to this thesis. These challenges and difficulties enforced us to change our plans and research directions many times during the PhD journey, in order to reach our final target. We learned a lot from these challenges and difficulties, not just in the scientific level, but also in the personal level in our life. The main lesson that we learned in our personal life is that the dreams are not always easy to be realized, and many obstacles may face the person during his work, and this requires him to never lose the hope and the trust in himself, as long as he works hard to achieve his objectives, and believes that Allah not to waste the wage of whoever does a good work.

Appendix A

The Implementation Classes

In this appendix, we present our implementations of the proposed RTSJ classes of the patterns, components and some examples presented in the thesis. These classes are a prototype implementation that have been tested only on a certain cases, scenarios, and examples, and some details of error-handling may be ignored or removed at some places to ease the reading of the classs. Also, these classes were tested only on the JRate implementation of the RTSJ, but it is not tested on any other implementations.

A.1 The Component Model

```
package RTCOM;
import javax.realtime.*;
import java.io.*;
import java.util.*;
/*////////////////Component Class//////////////////
Function:Represents the Component Model
*/////////////////////////////////////////////////
public class ComponentCls
 implements IComponent {
 IMemoryModelControler theMemoryControler; //The memory contoller sub
comp.
 IObjectAllocator allocator;               //The reusable objects
allocator sub comp
 public IContainer theContainer = null;    //The container
 public IMemoryModel theMemModel;          //The memory model
 IMemoryAllocator theMemoryAllocator;      //The memory allocator
 java.util.HashMap theHandlersPools;       //Poll of handlers
 java.util.HashMap theTasksPool;           //Pool of inner tasks
 /*///////////////getMemModel() Method///////////////
 Function:Retrieve the memory model of the Componeent
 Parameters->None
 */
 ////////////////////////////////////////////////////
 public IMemoryModel getMemModel() {
  if (theMemModel == null) {
   System.out.println("No Memory Model");
   return null;
  }
  return theMemModel;
 }
 /*///////////////start() Method//////////////////////
 Function:starts the execution of the driver thread of
     this component
 Parameters->None
 *///////////////////////////////////////////////////
 public void start() {
  Iterator iterator = theTasksPool.keySet().iterator();

  while (iterator.hasNext()) {
   String sname = (String)iterator.next();              //get the
name of next task
   Schedulable so = (Schedulable)theTasksPool.get(sname); //get the
```

```
next task

   if (so instanceof RealtimeThread) {
    RealtimeThread r = (RealtimeThread)so;
    r.start(); //start the execution of the task
   }
  }
 }
 /*///////////setContainer() Method///////////////////
 Function:Assign the container of this component
 Parameters->        container:A reference to the container
 *////////////////////////////////////////////////////
 public void setContainer(IContainer container) {
  theContainer = container;
 }
 /*////////////getContainer() Method//////////////////
 Function:Retrieve the container of this component
 Parameters->None
 */
 ////////////////////////////////////////////////////
 public IContainer getContainer() {
  return theContainer;
 }
 /*/////////////setBinder() Method///////////////////
 Function:Assign the binder object of this component
 Parameters->        binbder:The ninding object
 *////////////////////////////////////////////////////
 public void setBinder(IBinding binder) { }
 /*/////////////setMemoryModel() Method///////////////
 Function:Assigne the memory model component
 Parameters->memModel:The memory model
   *////////////////////////////////////////////////////
    public void setMemoryModel(IMemoryModel memModel) { }
    /*/////////getComName() Method//////////////////////
    Function:Retrieve the name of this componenyt
    Parameters->None
    *////////////////////////////////////////////////////
    public String getComName() {
     return comName;
    }
    /*//////////////setComName() Method/////////////////
    Function:Assgn a name for this component
    Parameters-> name:The new name of the component
    *////////////////////////////////////////////////////
    public void setComName(String name) {
     comName = name;
    }
    /*////////////addSMATask() Method//////////////////
    Function:Adds a new SMA's task to the memory model
        of the component
    Parameters->        taskName:The name of the task
                    SchedulableExecutor:The executor class
                    schedulingP:Task's scheduling parameters
                    releaseP:Task's release parameters
                    scopeSize:The size of the scoped memory
                    immortalSize:The size of immortal memory
                    mareaType:The type of scoped memory
                    group:processib=ng groyp paramters
                    StackLogicCls:The stack logic component
                        assigned to this task
    *////////////////////////////////////////////////////
    public ReusableRunnableStack addSMATask(String taskName, Class
   SchedulableExecutor, SchedulingParameters schedulingP,
        ReleaseParameters releaseP, long scopeSize, long immortalSize,
   Class mareaType, ProcessingGroupParameters group,
        Class StackLogicCls) {
     ReusableRunnableStack rs;
     MemoryParameters memP = new MemoryParameters(scopeSize,
   immortalSize); //Create the memory parameters

     if (!IStackLogic.class.isAssignableFrom(StackLogicCls)) {
      System.out.println("error creating the task: the logic does not
   extend the ISTackLogic interface");
      return null;
     }
     IStackLogic stacklogic = null;

     try {
      stacklogic = (IStackLogic)StackLogicCls.newInstance(); //creTe
   the logic sub-comp
     } catch (Exception ie) {
      System.out.println("Problem creating the Logic Object");
     }

     if (SchedulableExecutor.isAssignableFrom(RealtimeThread.class)) {
      ForkedMemoryModel memModel = (ForkedMemoryModel)getMemModel();
   //get the memory model
      ScopedMemory marea = memModel.attachNewSMA(mareaType, scopeSize,
   scopeSize); //should be in the same memory area
      rs = new ReusableRunnableStack(); //create the RRS
      rs.setStackLogic(stacklogic); //assign the stack logic to be
   executed by the RSS
      RealtimeThread rtThread = new RealtimeThread(schedulingP,
   releaseP, memP, (MemoryArea)getContainerMA(), group,
          rs); //The driving thread of the task
      addTask(taskName, rtThread);
   //add the task to the component
     }
     else if
   (SchedulableExecutor.isAssignableFrom(NoHeapRealtimeThread.class)) {
   //not implemented
     }
     return rs;
    }

    /*///////////addPeriodicSMATask() Method/////////////
    Function:Adds a periodic SMA's task to this omponent
    Parameters->        taskName:The name of the task
                    SchedulableExecutor:The executor class
                    schedulingP:Task's scheduling parameters
                    releaseP:Task's release parameters
                    scopeSize:The size of the scoped memory
                    immortalSize:The size of immortal memory
                    mareaType:The type of scoped memory
                    group:processib=ng groyp paramters
                    StackLogicCls:The stack logic component
                        assigned to this task
    *///////////////////////////////////////////////////
    public PeriodicReusableRunnableStack addPeriodicSMATask(String
   taskName, Class SchedulableExecutor,
        SchedulingParameters schedulingP, PeriodicParameters releaseP,
   long scopeSize, long immortalSize, Class mareaType,
        ProcessingGroupParameters group, Class StackLogicCls) {
     PeriodicReusableRunnableStack rs;
   // RRS pattern
     MemoryParameters memP = new MemoryParameters(scopeSize,
   immortalSize); //create the thread memory aparameters

     if (!IStackLogic.class.isAssignableFrom(StackLogicCls))
   //We assume only stack logic component
     {
      System.out.println("error creating the task: the logic does not
   extend the ISTackLogic interface");
      return null;
     }
     IStackLogic stacklogic = null;
     try {
      stacklogic = (IStackLogic)StackLogicCls.newInstance(); //create
   an instance of the stack logic sub-component
     } catch (Exception ie) {
      System.out.println("Problem creating the Logic Object");
     }

     if (SchedulableExecutor.isAssignableFrom(RealtimeThread.class)) {
      ForkedMemoryModel memModel =
   (ForkedMemoryModel)getMemModel();//Get the assigned memory model
      ScopedMemory marea = memModel.attachNewSMA(mareaType, scopeSize,
   scopeSize); //should be in the same memory area//CMA
      rs = new PeriodicReusableRunnableStack();
   //create the periodic RRS instance
      rs.setStackLogic(stacklogic); //assign the stack logic to be
   executed by the the periodic RRS
      PeriodicRealtimeThread rtThread =
          new PeriodicRealtimeThread(schedulingP, null, memP,
   (MemoryArea)getContainerMA(), group,
              rs); //create the periodic executing thread
      rtThread.setReleaseParameters(Times.sender_start_pm,
   Times.sender_start_pn, Times.sender_start_h,
          Times.sender_start_m, Times.sender_start_mi,
   Times.sender_start_ns); //we use fixed values here for testing
      addTask(taskName, rtThread);
   //add the task to the component
     }
     else if
   (SchedulableExecutor.isAssignableFrom(NoHeapRealtimeThread.class)) {
   }
     return rs;
    }
    /*///////////////addSMATask() Method/////////////////
    Function:Adds a periodic SMA's task to this omponent
    Parameters->        taskName:The name of the task
                    SchedulableExecutor:The executor class
                    schedulingP:Task's scheduling parameters
                    releaseP:Task's release parameters
                    memArea:The scoped memory of the task
                    group:processib=ng groyp paramters
                    logic:The stack logic component
                        assigned to this task
    */
    ////////////////////////////////////////////////////
    public ReusableRunnableStack addSMATask(String taskName, Class
   SchedulableExecutor, SchedulingParameters schedulingP,
        ReleaseParameters releaseP, ScopedMemory memArea,
   ProcessingGroupParameters group, IStackLogic logic) {
     ReusableRunnableStack rs = null;
     MemoryParameters memP =
         new MemoryParameters(memArea.getMaximumSize(),
   memArea.getMaximumSize()); //not sure about maxSize

     if (SchedulableExecutor.isAssignableFrom(RealtimeThread.class)) {
      //addtoTasks
      IMemoryModel memModel = getMemModel(); //get the memory model
      rs = new ReusableRunnableStack();       //create the RRS
      rs.setStackLogic(logic);                //assign the stack logic
   of the RRS
      RealtimeThread rtThread = new RealtimeThread(schedulingP,
   releaseP, memP, (MemoryArea)getContainerMA(), group,
          rs); //create the driving thread of the component
     //satart should be called from initilizeComponents()
     }
     else if
   (SchedulableExecutor.isAssignableFrom(NoHeapRealtimeThread.class)) {
   }
```

```
 return rs;
}
/*///////////////getMemoryController() Method/////////
Function:Retrieve the memory life time controller
of this component
Parameters->None
*////////////////////////////////////////////////////
public IMemoryModelControler getMemoryController() {
 return null;
}
/*//////////////setMemoryAllocator() Method///////////
Function:Assigns the memory life time controller
Parameters->memControllerCls:The class name of the
    memory life time controller
*////////////////////////////////////////////////////
public void setMemoryController(final Class memControllerCls) {
 if
(memControllerCls.isAssignableFrom(IMemoryModelControler.class)) {
//creates the memory controller in the CMA
   ((ScopedMemory)getCMA()).enter(new Runnable() {
    public void run() {
     getMemModel().setMemoryController(memControllerCls);
    }
   });
  }
}
/*////////////setMemoryAllocator() Method/////////////
Function:Assigna a memory allocator
Parameters->memoryAllocator:Memory allocator component
*////////////////////////////////////////////////////
public void setMemoryAllocator(IMemoryAllocator memoryAllocator) {
 theMemoryAllocator = memoryAllocator;
}
/*////////////getMemoryAllocator() Method/////////////
Function:Retrieve th memory allocator component
Parameters->None
*////////////////////////////////////////////////////
public IMemoryAllocator getMemoryAllocator() {
 return theMemoryAllocator;
}

/*////////////////getTask() Method///////////////////
Function:Retrieves a running task from the componnet
Parameters->TaskNAme:The name of the required task
*////////////////////////////////////////////////////
public Schedulable getTask(String TaskName) {
 return theTasksPool.get(TaskName);
}
/*//////////////addTask() Method/////////////////////
Function:Adds a task to the component
Parameters->        TaskNAme:The name of the task
                task:The new task
*////////////////////////////////////////////////////
public void addTask(String TaskName, Schedulable task) {
 theTasksPool.put(TaskName, task);
}
/*/////////getHandlerPool() Method///////////////////
Function:Retrieves the handlers pool
Parameters->The name of the target pool
*////////////////////////////////////////////////////
public IHPool getHandlerPool(String poolName) {
 return theHandlersPools.get(poolName);
}
/*////////////addHandlerPool() Method////////////////
Function:Adds a handler pool sub-component to this comp.
Parameters->        poolName:The name of the added pool
                pool:The pool component
*////////////////////////////////////////////////////
```

```
 public void addHandlerPool(String poolName, IHPool pool) {
  theHandlersPools.put(poolName, pool);
 }
 /*/////////setMemoryControler() Method////////////////
 Function:Sets the memory lifetime sub-comp of
     this component
 Parameters->memoryController:The added memory sub-component
 *///////////////////////////////////////////////////
 public void setMemoryControler(IMemoryModelControler
memoryControler) {
  theMemoryControler = memoryControler;
 }
 /*////////////getMemoryControler() Method/////////////
 Function:Gets the MemoryController of this component
 Parameters->None
 *///////////////////////////////////////////////////
 public IMemoryModelControler getMemoryControler() {
  return theMemoryControler;
 }
 /*//////////////Constructor/////////////////////////
 Function:Constructor
 Parameters->None
 *///////////////////////////////////////////////////
 public ComponentCls() {
  theHandlersPools = new java.util.HashMap();
  theTasksPool = new java.util.HashMap();
 }
 /*///////////////setMemoryControler() Method//////////
 Function:Assigne the memory life time controller
     sub-comp of this component
 Parameters->        memoryControlerCls: The class name of
     the added sub-comp
 *//////////////////////////////////////////////////
 public void setMemoryControler(Class memoryControlerCls) {
  if (memoryControlerCls.isAssignableFrom
(IMemoryModelControler.class)) {
   try {
    theMemoryControler =
(IMemoryModelControler)((ScopedMemory)theMemModel.getCMA()).newInsta
nce(memoryControlerCls);
   } catch (Exception ex) { }
  }
 }

 /*//////////////setObjectAllocator() Method///////////
 Function:Assigna an objecgt allocator sub-como of
     this component
 Parameters->        memoryControlerCls:The added subcomponent
 *//////////////////////////////////////////////////
 public void setObjectAllocator(Class memoryControlerCls) {
  if (memoryControlerCls.isAssignableFrom
(IMemoryModelControler.class)) {
   try {
    theMemoryControler =
(IMemoryModelControler)((ScopedMemory)theMemModel.getCMA()).newInsta
nce(memoryControlerCls);
   } catch (Exception ex) { }
  }
 }
 String comName = "";
 /*///////////////Constructor/////////////////////////
 Function:Initializes the component name
 Parameters-> compName:The given name to the component
 *///////////////////////////////////////////////////
 public ComponentCls(String compName) {
  theHandlersPools = new java.util.HashMap();
  theTasksPool = new java.util.HashMap();
  comName = compName;
 }
 /*///////////////Constructor/////////////////////////
 Function:Initializes the component
 Parameters->container:The enclosing container
             compName:The name of the component
 *///////////////////////////////////////////////////
 public ComponentCls(IContainer container, String compName) {
  theHandlersPools = new java.util.HashMap();
  theTasksPool = new java.util.HashMap();
  theContainer = container;
  theContainer.addComponent(this);
  comName = compName;
 }
 /*///////////////init() Method///////////////////////
 Function:initialize the component
 Parameters->memModel:The memory model of the component
 *////////////////////////////////////////////////////
 public void init(IMemoryModel memModel) {
  theMemModel = memModel;
 }
 /*///////////////getCMA() Method///////////////////////
 Function:Get the CMA of this component
 Parameters->None
 *////////////////////////////////////////////////////
 public MemoryArea getCMA() {
  return theMemModel.getCMA();
 }
 /*/////////////getContainerMA() Method////////////////
 Function:Get the container memory area of
     this component
 Parameters->None
 *////////////////////////////////////////////////////
 public MemoryArea getContainerMA() {
  return theMemModel.getContainerMA();
 }
 /*//////////////init() Method///////////////////////
 Function: Initialiize the component
 Parameters->
 *//////////////////////////////////////////////////
 public void init(Class memCls, long initSize, long MaxSize, int
LIMIT, Class SMAClass, long [] initialSizes,
     long [] maxSizes, Class memCtrlCls) {
  final ForkedMemoryModel fmm = new ForkedMemoryModel(); //create
thememory model

  if (memCls.isAssignableFrom(LTMemory.class)) {         //build the
structure of the memory model
   fmm.buildForkedMemory(memCls, initSize, MaxSize, LIMIT, SMAClass,
initialSizes, maxSizes, NamedObjectFastMap.class,
       Queue.class, GeneralObjectAllocatorCls.class, memCtrlCls);
  }
  theMemModel = (IMemoryModel)fmm;
  //the allocator
  ((LTMemory)fmm.getCMA()).enter(new Runnable() {
   public void run() //this runnable should be by the container
allocator
   {//create the objects allocator in the CMA
fmm.setObjectAllocator(GeneralObjectAllocatorCls.instance());
   }
  });
 }

 /*//////////////init() Method///////////////////////
 Function: Initialiize the component
 Parameters-># 	theCMA:The comoon memory area
                 LIMIT: The limit
                 SMAClass:The clsas of memory area
                 initialSizes:The initial sizes of SMAs
                 maxSizes:The max sizes of SMAs
                 memCtrlCls :memory lifetime controller Class
 *///////////////////////////////////////////////////
 public void init(LTMemory theCMA, int LIMIT, Class SMAClass, long
[] initialSizes, long [] maxSizes, Class memCtrlCls)
     {
  final ForkedMemoryModel fmm = new ForkedMemoryModel(); //create
the memory model                                //build thes
structure of the memory model
   fmm.buildForkedMemory(theCMA, LIMIT, SMAClass, initialSizes,
maxSizes, NamedObjectFastMap.class, Queue.class,
       GeneralObjectAllocatorCls.class, memCtrlCls);
  theMemModel = (IMemoryModel)fmm; //assign the memory model
  GeneralObjectAllocatorCls gallocator;

  try {
   gallocator = new GeneralObjectAllocatorCls(); //create the
allocator
  } catch (Exception e) { }
  fmm.setObjectAllocator(gallocator);       ////assign the allocator
 }
}
```

A.2 The Container Model

```
package RTCOM;
import javax.realtime.*;
import java.util.*;

/*/////////////////////////////////////////////////////////
Function: The class of the Container Component
*///////////////////////////////////////////////////////
public abstract class ContainerCls
 implements IContainer {
 HashMap theComponents;     //The components
 ScopedMemory containerMA; //Container memory area
 /*/////////////////////////////////////////////////////////
 Function:Retrieves the memory area of the container
 ParametersNone:None
 *///////////////////////////////////////////////////////
 public ScopedMemory getMemArea() {
  return containerMA;
 }
 /*/////////////////////////////////////////////////////////
 Function:creates the container memory area
 Parameters:None
 *///////////////////////////////////////////////////////
 public ContainerCls() {
  containerMA = (ScopedMemory)RealtimeThread.getCurrentMemoryArea();
 }
 /*/////////////////////////////////////////////////////////
 Function:Initialiize the sontainer
 Parameters:        ma:Memory area
 *///////////////////////////////////////////////////////
 public ContainerCls(MemoryArea ma) {
  containerMA = (ScopedMemory)ma;
  theComponents = new HashMap();
 }

 /*/////////////////////////////////////////////////////////
 Function:Initialiize the container object
 Parameters:        MemType:Memory area
            initSize:initial size of memory
            maxSize:Max size of memory
 *///////////////////////////////////////////////////////
 public ContainerCls(Class MemType, long initSize, long maxSize) {
  if (MemType == LTMemory.class) {
   containerMA = new LTMemory(initSize, maxSize);
  } else {
   System.out.println("Not Implemented");
  }
  theComponents = new HashMap();
 }
 /*/////////////////////////////////////////////////////////
 Function:Retrieve the components enclosed in this container
 Parameters:None
 *///////////////////////////////////////////////////////
 public HashMap getComponents() {
  return theComponents;
 }

 /*/////////////////////////////////////////////////////////
 Function:adds a component to the container
 Parameters->        com:The added component
 *///////////////////////////////////////////////////////
 public void addComponent(IComponent com) {
  com.setContainer(this); //set this comp as a container for the added
component

  if (theComponents == null)
```

```
   theComponents = new java.util.HashMap();
  theComponents.put(com.getComName(), com); //add the component to the
component list within this container
 }
 /*//////////////////////////////////////////////////////////
 Function:Get a component by its name from the container
 Parameters:         ComName:The name of the required component
 *///////////////////////////////////////////////////////
 public IComponent getComponent(String ComName) {
  return (IComponent)theComponents.get(ComName);
 }

 /*//////////////////////////////////////////////////////////
 Function:initializes the container
 Parameters->         initSize:initial size of the container memory
                 maxSize: max size of the container memory
 *///////////////////////////////////////////////////////
 public void initialize(long initSize, long maxSize) {
  LTMemory Initial = new LTMemory(initSize, maxSize);
  //create the driving thread of the container
  final RealtimeThread T = new RealtimeThread(null, null, null,
Initial, null, new Runnable() {
   public void run() {
    BuildComponents(); //build the inner components of this container

    synchronized (ContainerCls.class) {
     try {
      ContainerCls.class.wait();
     } catch (Exception zz) { }
    }
    ;
   }
  });
  T.start(); //start the container execution

  synchronized (ContainerCls.class) {
   try {
    ContainerCls.class.wait();
   } catch (Exception zz) { }
  }
  ;
 }
}
```

A.3 The Forked Memory Model Pattern

```
package RTCOM;
import javax.realtime.*;
 /*///////////////ForkedMemoryModel Class/////////////////////
 Function: A class to represent the Forked Memory Pattern,
 in which there is a contMa,CMA, and a set of SMSs
 *//////////////////////////////////////////////////////////
public class ForkedMemoryModel
 implements IMemoryModel {
 IMemoryModelControler theMemoryControler;//memory Lifetime controller
of the SMSs
 IObjectAllocator ContainerObjAllocator = null; //reusable object
allocator
 ScopedMemory CMA;//The common memory area
 MemoryArea ContMA;//The Container Memory Area
 int maxSMAs;//The max number of SMAs
 int curSMAs = 0;//The Total current SMAs
 BuildSMAsMethod buildSMAsMeth = new BuildSMAsMethod();//create an
instance of the builder class to use it as an encapsulated method
 BuildSMAsMethod.AttachNewSMAMethodCls newAttachedSMA = new
BuildSMAsMethod.AttachNewSMAMethodCls();//create an instance of a class
that is used as an encapsulated method for Attaching SMA
 IComponent ICOM = null;
 IQueue smaq; //to be accessible within the runnable

 /*//////createContainerObjectAllocator() Method//////////
Function:Create Reusable objects allocator within the container, this
method has to be called from the containerMA
Paramters->None
 *//////////////////////////////////////////////////////////
 public void createContainerObjectAllocator() {
  ContainerObjAllocator = GeneralObjectAllocatorCls.instance();//create
an instance of the allocator
 }

 /*////////getContainerObjectAllocator() Method///////////
Function:retrieve the object allocator from the allocator
Paramters->None
 *//////////////////////////////////////////////////////////
 public IObjectAllocator getContainerObjectAllocator() {
  return ContainerObjAllocator;
 }

 /*////////////getMemoryController() Method//////////////
Function:Retrieve the memory life time controller associated with thuis
pattern
Paramters->None
 *//////////////////////////////////////////////////////////
 public IMemoryModelControler getMemoryController() {
  return theMemoryControler;
 }

 /*///////////////setSMAsQueue() Method//////////////////
Function:Assign a Queue component of the SMAs of the model
Paramters->        que:The Queue of SMSa
 *//////////////////////////////////////////////////////////
 public void setSMAsQueue(IQueue que) {
  getNamedPortal().insertObject("SMAsQueue", que);//save a reference of
the Queue in the MNPortal of the common memory area
 }

 /*///////////////getSMAsQueue() Method//////////////////
Function:Retrieve the Queue that holds all the SMAs
Paramters->None
 *//////////////////////////////////////////////////////////
 public IQueue getSMAsQueue() {
  return getNamedPortal().getObject("SMAsQueue");
 }

 /*///////////////getContainerMA() Method/////////////////
Function:Retrieve the Container Memory area
Paramters-> None
 *///////////////////////////////////////////////////////
 public ScopedMemory getContainerMA() {
  return ContMA;
 }

 /*/////////////getObjectAllocator() Method///////////////
Function:Retrieve the object allocataor
Paramters->None
 *///////////////////////////////////////////////////////
 public IObjectAllocator getObjectAllocator() {
         //Retrieve the allocator from the MNOPortal of the CMA
  return
((INamedObjectCollection)(((LTMemory)getCMA()).getPortal())).getObject(
"allocator");
 }

 /*//////////////////getCMA() Method//////////////////////
Function:Retrieves the common memory area
Paramters->None
 *///////////////////////////////////////////////////////
 public Object getCMA() //this can be heap,immortal,or scoped
 {
  return CMA;
 }

 /*///////////////getPortalOfCMA() Method/////////////////
Function:Retrieve the Portal of the CMA
Paramters->None
 *///////////////////////////////////////////////////////
 public Object
     getPortalOfCMA() //returns the MNOP portal, it can be the portal
of the CMA if it is scoped or user  created portal
 {
  return CMA.getPortal();
 }

 /*//////////////getSMAsScopes() Method///////////////////
Function:Retrieve the Qqueue that holds the SMAs
Paramters->None
 *///////////////////////////////////////////////////////
 public IQueue getSMAsScopes() {
  return getNamedPortal().getObject("SMAsQueue");
 }

 /*//////////////setNamedPortal() Method//////////////////
Function:Assign a named collection obj as a MNOPortal of the CMA
Paramters-> namedCollectionPortal:The Multi Named-Object collection to
hold the shared objects in the portal
 *///////////////////////////////////////////////////////
 public void setNamedPortal(INamedObjectCollection
namedCollectionPortal) {
  if (CMA instanceof ScopedMemory) {
   CMA.setPortal(namedCollectionPortal);//assign the MNOP portal
  }
 }

 /*////////////getNamedPortal() Method////////////////////
Function:Retrieve the named MNOPortal of the CMA
Paramters->None
 *///////////////////////////////////////////////////////
 public INamedObjectCollection getNamedPortal() {
         //u have to be in the cma or upper to access it
  return (INamedObjectCollection)CMA.getPortal();//return the MNOP
portal
 }

 /*////////setMemoryController() Method///////////////////
Function:Assign a memory lifetime controller for the FFM
Paramters-> MemoryCtrlCls:The class of the required liftime controller
type, e.g. DualFork
 *////////////////////////////////////////////////////////
 public void setMemoryController(final Class MemoryCtrlCls) {
  try {
   IMemoryModelControler memCtrlr;//reference to the memory ife time
controller

   if (MemoryCtrlCls.isAssignableFrom(DualFork.class)) {//the case of
using the default dual fork as memory life time controller
    IQueue SMAsQueue =
((IQueue)getNamedPortal().getObject("SMAsQueue"));//Get the SMAs Queue
from the portal
    memCtrlr = new DualFork(CMA, SMAsQueue);//Generate an instance of
the lifetime controller
   } else {
    memCtrlr =
(IMemoryModelControler)MemoryCtrlCls.newInstance();//Create an instance
of the given type of the life time controller
   }

ForkedMemoryModel.this.getNamedPortal().insertObject("memController",
memCtrlr);//add the life time cointroller to the MNOP portal
   theMemoryControler = memCtrlr;//update the class by the lifetime
controller
  } catch (Exception e) {
   System.out.println("Error......");
  }
 }

 /*///////////////attachNewSMA() Method//////////////////
Function:Attaches a new SMA to the FMM
Paramters->
 *////////////////////////////////////////////////////////
 public ScopedMemory attachNewSMA(final Class reqMemType, final long
initSize, final long maxSize)
     { //we may set name for it to save it and reaccess it later
  //should be the containerMEMAREA
  long [] x = { 0 };//create a single element array
  long [] y = { 0 };//create a single element array
  x[0] = initSize;//prepare the initial sizes array with a single value
for all SMA
  y[0] = maxSize;//prepare the max sizes array with a single value for
all SMA
  //assign the arguments of the buildSMAsMeth encapsulated method
  ForkedMemoryModel.this.buildSMAsMeth.setArguments(new Object[]
      {x, y, "SMAsQueue", reqMemType, null});//Assign the arguments of
the encapsulated method in buildSMAsMeth
  //assign the arguments of the newAttachedSMA encapsulated method
  ForkedMemoryModel.this.newAttachedSMA.setArguments(new Object[]
      {x, y, "SMAsQueue", reqMemType, null, this});//Assign the
arguments of the encapsulated method in newAttachedSMA
  ((MemoryArea)getCMA()).enter(buildSMAsMeth);//execute the method
encapsulated in buildSMAsMeth
  ++curSMAs;//increment the current SMAs count
  return ForkedMemoryModel.this.newAttachedSMA.createdMemory;
 }
 static RealtimeThread tlocker;

 final String x = "Locker";
 DualFork.WedgeThread wedge;///????
 /*//////////////////////////////////////////////////////////
Function:Build The Fork Memory
Paramters-> typeOfCMA:
             CMA_InitialSize:initial Size of the CMA
             CMA_ MaxSize:max size of the CMA
             maxSubStacks:maxi number of sun Stacks
             typeOfSMA:the required scoped memory class
             SMA_ InitialSizes[]:the initial sizes of the SMAs
             SMAs_MaxSizes[]:the max sizes of SMAs
             MPortalType: the required class type of the MNOP
             theSMAQueue:The class of the Queue to hold the SMAs
             ObjAllocator:The class of the reusable obj allocator
             memCtrlCls:the scoped memory class type of the SMA
 *//////////////////////////////////////////////////////////
 public void buildForkedMemory(final Class typeOfCMA, final long
CMA_InitialSize, final long CMA_MaxSize,
     final int maxSubStacks, final Class typeOfSMA, final long []
SMAs_InitialSizes, final long [] SMAs_MaxSizes,
     final Class MPortalType, final Class theSMAQueue, final Class
ObjAllocator, final Class memCtrlCls) { //working
//we can use it in an example of n parallel static[i.e. has no reusable
threads] tasks, e.g. periodic realtime threads, are executing on this
model and they can use the shared memory for communication, and in the
same example we can use the fork thread for keeping some mem areas
alive to do do communication
  maxSMAs = maxSubStacks;//assign the maximum allowed number of SMAs
  if (typeOfCMA == LTMemory.class) {
   CMA = new LTMemory(CMA_InitialSize, CMA_MaxSize);//create the common
memory
  } else if (typeOfCMA == VTMemory.class) {
   CMA = new VTMemory(CMA_InitialSize, CMA_MaxSize);//create the common
memory
  }
  else if (typeOfCMA == HeapMemory.class) { //Not implemented}
  final ScopedMemory fCMA = CMA;
  RealtimeThread currentRTThread =
RealtimeThread.currentRealtimeThread();//Get the current thread
  ContMA = currentRTThread.getCurrentMemoryArea();//Get the current
Memory area
  tlocker = new RealtimeThread(null, null, null, ContMA, null, new
Runnable() {
   public void run() {
    try {
ForkedMemoryModel.this.setNamedPortal((INamedObjectCollection)MPortalTy
pe.newInstance());//create an MNOP portal instance and save it as the
portal of this FMM
     IQueue smaq = (IQueue)theSMAQueue.newInstance();//create the SMAs
Queue
     setSMAsQueue(smaq);//Save the SMAs Queue in the portal
    } catch (Exception e) {
     System.out.println("Exception......"+e);
    }

    if (typeOfSMA != null) {
     curSMAs = SMAs_InitialSizes.length;//Initialize the count of
current SMAs

     if (typeOfSMA == LTMemory.class) {
      for (int i = 0; i < SMAs_InitialSizes.length; i++) {//create the
required number of the SMAs and save them in the SMAs Queue
((IQueue)getNamedPortal().getObject("SMAsQueue")).insert(new
LTMemory(SMAs_InitialSizes[i], SMAs_MaxSizes[i]),
           i);
      }
     } else if (typeOfCMA == VTMemory.class) {
      for (int i = 0; i < SMAs_InitialSizes.length; i++) {//create the
required number of the SMAs and save them in the SMAs Queue
((IQueue)getNamedPortal().getObject("SMAsQueue")).insert(new
VTMemory(SMAs_InitialSizes[i], SMAs_MaxSizes[i]),
           i);
      }
     }
    }
    x.notifyAll();

    synchronized (this) { //has to be here to keep the mem structure
     try {
      wait();//?/
     } catch (Exception e) { }
    }
   }
  });
  tlocker.start();//run the thread that creates the pattern

  synchronized (x) { //has to be here to keep the mem structure
   try {
    x.wait();//??
   } catch (Exception e) { }
  }
 }

 /*/////////////////////////////////////////////////////////
Function:This method is used to build the structure of the Forked
Memory Model pattern
Paramters-> typeOfCMA:the required scoped memory class
            CMA_InitialSize:the initial size of the CMA
                        CMA_MaxSize//The max size of the CMA
            maxSubStacks:The MAx number of the sub stacks
            MPortalType:The class type of the portal
            theSMAQueue:The SMAs Queue that will hold the SMAs
            ObjAllocator:The class of the required reusable
                        object allocator
            memCtrlCls:The required class of the memory life
                        time controller
 *//////////////////////////////////////////////////////////
 public void buildForkedMemory(final Class typeOfCMA, final long
CMA_InitialSize, final long CMA_MaxSize,
     final int maxSubStacks, final Class MPortalType, final Class
theSMAQueue, final Class ObjAllocator,
     final Class memCtrlCls) {
//we can use it in an example of n parallel dynamic case, where the
SMAsQueue will be filled dynamically, or by the reusable runnablee
stack
  RealtimeThread currentRTThread =
RealtimeThread.currentRealtimeThread();//get the current thread
  ContMA = currentRTThread.getCurrentMemoryArea();//get the container
memory area
  maxSMAs = maxSubStacks;//assign the max n umber of the SMAs

  if (typeOfCMA == LTMemory.class) {
   CMA = new LTMemory(CMA_InitialSize, CMA_MaxSize);//Create the CMA
  } else if (typeOfCMA == VTMemory.class) {
   CMA = new VTMemory(CMA_InitialSize, CMA_MaxSize);//Create the CMA
  }
  else if (typeOfCMA == HeapMemory.class) { //Not implemented}
  /////////////////SMA-BUILDING////////////////
  CMA.enter(new Runnable() {
 /*/////////////////////////////////////////////////////////
#Function:This function has the logic of the encapsulated method of
this class
Paramters->None
 *//////////////////////////////////////////////////////////
   public void run() {
    try {
ForkedMemoryModel.this.setNamedPortal((INamedObjectCollection)MPortalTy
pe.newInstance());//assign the required MNOP portal
```

```
     setMemoryController(memCtrlCls);//Assign the given memory liftime
controller
     IQueue smaq = (IQueue)theSMAQueue.newInstance();//create an
instance of the SMAs Queue to hold the SMAs
     setSMAsQueue(smaq);//assign the created queue
    } catch (Exception e) {
     System.out.println("Error......"+e);
    }
   }
  });
 }

 /*///////////////buildForkedMemory() Method//////////////
Function:Build the forked memory model
Paramters-> theCMA:The Common memory area
            maxSubStacks:The count of the SubStacks
            typeOfSMA:The required scoped memory area type
            SMAs_InitialSizes:The initial sizes of the SMAs
            SMAs_MaxSizes:The max sizes of the SMAs
            MPortalType:The type of the portal
            theSMAsQueue:The SMAs Queu that hold the SMAs
            ObjAllocator:The class to be used to
                    create the Reusable object allocator
            memCtrlCls:the class to be used to create the
                    memory life time controller
 *////////////////////////////////////////////////////
 public void buildForkedMemory(MemoryArea theCMA, final int
maxSubStacks, Class typeOfSMA,
     final long [] SMAs_InitialSizes, final long [] SMAs_MaxSizes,
Class MPortalType, Class theSMAsQueue,
     Class ObjAllocator, final Class memCtrlCls) {
      if (theCMA instanceof LTMemory) {
   CMA = (LTMemory)theCMA;//assign the common memory area of FMM
   typeOfSMA = LTMemory.class;//Set the type's class
  } else if (theCMA instanceof VTMemory){
   CMA = (VTMemory)theCMA;//assign the common memory area of FMM
   typeOfSMA = VTMemory.class;//set the type's class
  }
  maxSMAs = maxSubStacks;//assign the Max number of SMAs
  try {
   smaq = (IQueue)theSMAsQueue.newInstance();//create the Queue of the
SMAs

   final INamedObjectCollection portal = null;
   RealtimeThread rtThread =
RealtimeThread.currentRealtimeThread();//Retreive the current Thread
   ContMA = (ScopedMemory)rtThread.getCurrentMemoryArea();//assign the
container memory area of FMM
   WedgeThread wedge = WedgeThread.startNewInstance(10,
ContMA);//create a wedge thread
   wedge.lockMA(tCMA);//Hold the CMA memory area
     }
  } catch (Exception ex) {
   System.out.println("Exception..,." + ex);
  }
//The following code executes the encapsulate method that builds the
SMAs of the FMM
  buildSMAsMeth.setArguments(new Object[]
      {SMAs_InitialSizes, SMAs_MaxSizes, smaq, typeOfSMA,
MPortalType});//assign the arguments of the encapsulated method
  CMA.enter(buildSMAsMeth);//execute the encapsulated method within the
CMA scoped memory area
  curSMAs = ((Integer)buildSMAsMeth.returnResult()).intValue();//Get
the count of the created SMAs as the return value of the encapsulated
method
  //The following code runs in the CMA to do some initializations
  CMA.enter(new Runnable() {
   public void run() {
```

```
ForkedMemoryModel.this.setNamedPortal((INamedObjectCollection)CMA.getPo
rtal());//get the portal
    ForkedMemoryModel.this.setSMAsQueue(
        ForkedMemoryModel.this.smaq); //assign the SMAS Queue
    ForkedMemoryModel.this.getNamedPortal().insertObject("wedgeThread",
wedge);//save the wedge thread reference in the MNOP portal
   }
  });
 }

  /*//////////BuildSMAsMethod Class///////////////////////
Function:An inner class that acts as an encapsulation method
for building the SMAs
 *///////////////////////////////////////////////////////
public class BuildSMAsMethod
  implements IEncapsulatedMethod {


 /*//////////CreateInitialSMAMethodCls Clsss//////////////
Function:An inner class that works as an encapsulated method
for building the initial SMAs
 *///////////////////////////////////////////////////////
  public class CreateInitialSMAMethodCls
   implements IEncapsulatedMethod {
   Class reqTypeOfMemory;
   boolean executed = false;

 /*////////////setArguments() Method/////////////////////
Function: Assign the value of the reqTypeOfMemory argument
Paramters-> arguments: an array that has the required value
 *///////////////////////////////////////////////////////
   public void setArguments(Object [] arguments) {
    reqTypeOfMemory = (Class)arguments[0];
   }

 /*//////////////returnResult() Method///////////////////
Function:As the method returns nothing, this method is
      a dummy
Paramters->None
 *///////////////////////////////////////////////////////
   public Object returnResult() {
    //not required so returns null
    return null;
   }

 /*/////////////////run() Logic/////////////////////////
Function:The method has the logic to initiate the SMAs
Paramters->None
 *///////////////////////////////////////////////////////
   public void run() {
    executed = true;//set the execution flag
    if (reqTypeOfMemory == LTMemory.class) {
     for (int i = 0; i < BuildSMAsMethod.this.SMAs_InitialSizes.length;
i++) {
//here SMAsQueue is in the container, the other option is to create it
in the CMA itself but this require passing its class instead of its
object, so it can be created within the CMA
      ScopedMemory MEM = new
LTMemory(BuildSMAsMethod.this.SMAs_InitialSizes[i],
BuildSMAsMethod.this.SMAs_MaxSizes[i]);//create the SMA

((Queue)(getNamedPortal().getObject(BuildSMAsMethod.this.SMAsQueue_Name
))).insert(MEM, i);//Save a ref to the SMA in the SMAs Queue


ForkedMemoryModel.this.getNamedPortal().insertObject(RealtimeThread.cur
rentRealtimeThread() + "." + i,
          RealtimeThread.currentRealtimeThread() + "." + i);//The a
reference of the current thread in the MNOP portal
     }
    }

    if (reqTypeOfMemory == VTMemory.class) {
     for (int i = 0; i < BuildSMAsMethod.this.SMAs_InitialSizes.length;
i++) {
                //not implemneted but Similar to LTMemory
     }
    }
   }
  } //end of Class CreateMethod

 /*//////////////AttachNewSMAMethodCls class//////////////
Function:An inner class used as an encapsulated method for
      attaching a new SMA to the Forked Memory
Paramters->None
 *//////////////////////////////////////////////////////////
  public class AttachNewSMAMethodCls
   implements IEncapsulatedMethod {
//The parameters of the method's arguments
   Class reqTypeOfMemory;//memory type of the new SMA
   long SMAInitSize;//initial size of the new memory area
   public int type = -1;
   long SMAMaxSize;//max size of the new memory area
   public Queue SMAsQueue;//Queue to hold the SMAs
   String SMAsQueue_Name;//A name for the new SMA
   boolean executed = false;// flag (executed or not)
   public ScopedMemory createdMemory = null;
   ForkedMemoryModel FMM;//reference to the containing forked memory
model

 /*///////////////setArguments() Method//////////////////
Function:Assigne the values to the encapsulated
      method arguments
Paramters->arguments:An array that has the required values
 *//////////////////////////////////////////////////////////
   public void setArguments(Object [] arguments) {
    if (arguments.length == 6) {
     FMM = (ForkedMemoryModel)arguments[5];//assign the parent FMM
argument
    }

    if (arguments[2]instanceof String) {
     type = 1;
     SMAsQueue_Name = (String)arguments[2];//assign the SMAs'Queue
argument
    } else {
     type = 2;
     SMAsQueue = (Queue)arguments[2];//assign the SMAsQueue ref.
    }
    reqTypeOfMemory = (Class)arguments[3];//assign the required memory
type of SMA
    SMAInitSize = ((long [])arguments[0])[0];//assign the initial sizes
of all SMAs
    SMAMaxSize = ((long [])arguments[1])[0];//assign the max sizes of
all SMAs
   }

 /*//////////////returnResult() Method///////////////////
Function:Return the Return value of the encapsulated method
Paramters->None
 *//////////////////////////////////////////////////////////
   public Object returnResult() {
    //Return the current SMAs
    return new Integer(BuildSMAsMethod.this.curSMAs);
   }
 /*/////////////////run() Method/////////////////////////
Function:Has the logic of the encapsulated method
Paramters->None
 *//////////////////////////////////////////////////////
   public void run() {
    RealtimeThread.getCurrentMemoryArea();//get the current thread
    executed = true;//set execution flag
    if (reqTypeOfMemory == LTMemory.class) {//in case of LTMemory
//here SMAsQueue is in the container, the other option is to create it
in the CMA itself but this require passing its class instead of its
object, so it can be created within the CMA
      createdMemory = new LTMemory(SMAInitSize, SMAMaxSize);//create
the scoped memory area
      RealtimeThread currentRTThread =
RealtimeThread.currentRealtimeThread();//get the current thread
      ScopedMemory MA =
(ScopedMemory)currentRTThread.getCurrentMemoryArea();//get the memory
area of the current thread
      long curSMAS = BuildSMAsMethod.this.curSMAs;//get the number of
the SMAs
//the next statement
      AttachNewSMAMethodCls.this.FMM.CMA.enter(new Runnable() {
       public void run() {
        if (type == 1) {
         RealtimeThread currentRTThread =
RealtimeThread.currentRealtimeThread();//get the current thread
         ScopedMemory ma =
(ScopedMemory)currentRTThread.getCurrentMemoryArea();//get the memory
area of the current thraed
         INamedObjectCollection gg =
             (INamedObjectCollection)(ma.getPortal());//get the
MNOPportal
         AttachNewSMAMethodCls.this.SMAsQueue
            = (Queue)FMM.getSMAsQueue(); //assign the SMAsQueuu
parameter
       }//end of if(type==1)
      }//end of run()
     });//end of enter(....
     SMAsQueue.insert(createdMemory, (int)curSMAS++);
    }//end of if (reqTypeOfMemory == LTMemory.class)
   }//end of run()
  } //end of Class Attach a new SMA

  boolean executed = false;//a flag of execution
  Queue SMAsQueue;//Queue of the SMAs
  String SMAsQueue_Name;//Name of the SMAs Queue
  long [] SMAs_InitialSizes;//initiall sizes of all SMAs
  long [] SMAs_MaxSizes;//max sizes of all SMAs
  Class thetypeOfMemory;//the type of the SMA
  int curSMAs;//total number of SMAs
  CreateInitialSMAMethodCls createInitSMAMethod = new
CreateInitialSMAMethodCls();//an instance of the encaqpsulated method
class

  Class portalType;//type of portal
  int type = -1;//type of method

 /*/////////////////setArguments() Method//////////////////
Function:Assign the arguments of this class that acts as an
       encapsulated method
Paramters-> arguments:An array of objects that holds
      the arguments
 *//////////////////////////////////////////////////////
  public void setArguments(Object [] arguments) {

   SMAs_InitialSizes = (long [])arguments[0];//initial size value of
all SMAs
```

```
   SMAs_MaxSizes = (long [])arguments[1];//Max Size value of all SMAs

   if (arguments[2]instanceof String) {
    SMAsQueue_Name = (String)arguments[2];//Assignt the name of the
queue
    type = 1;
   } else {
    SMAsQueue = (Queue)arguments[2];//assign the given SMAs Queue
object
    type = 2;
   }

   thetypeOfMemory = (Class)arguments[3];//The required scoped memory
type

   if (arguments.length > 4)
    portalType = (Class)arguments[4];//the required portal type
   else
    portalType = null;
  }

/*////////////////returnResult() Method////////////////
Function:return the result of the encapsulated method, which id
       the total current number of SMAs
Paramters-> None
 *////////////////////////////////////////////////////////
  public Object returnResult() {
   if (executed)//if already executed
    return new Integer(curSMAs);//return the calculated result
   else
    return null;//otherwise, return nothing
  }

 /*//////////////////////run() Method///////////////////
Function:runs the logic of the encapsulated method, to
       build the SMAs
Paramters->None
 *////////////////////////////////////////////////////////
  public void run() {
   INamedObjectCollection portal = null;//create a portal reference
   NamedObjectFastMap p = new NamedObjectFastMap();//create a
collection as MNOP
   RealtimeThread currentRTThread =
RealtimeThread.currentRealtimeThread();//get the current real time
thread
   ScopedMemory theCMA =
(ScopedMemory)currentRTThread.getCurrentMemoryArea();//retrieve the
memory area of the current thread

   try {
    if (portalType != null)
     portal =
(INamedObjectCollection)theCMA.newInstance(portalType);//create a new
portal of the given type, if no one exists
    else
     portal = new NamedObjectFastMap();//create a new portal of the
default type(NamedObjectFastMap), if no one exists
    theCMA.setPortal(portal);//assign the MNOP portal of the CMA
   } catch (Exception ex) {
   }
   executed = true;//set the executed flag
   curSMAs = SMAs_InitialSizes.length;//update the total number of the
current SMAs
   ScopedMemory sm=null;//create a scoped memory ref.

   if (type == 1) {
    sm = ForkedMemoryModel.this.getContainerMA();//use the container MA
   } else if (type == 2) {
    sm = (ScopedMemory)MemoryArea.getMemoryArea(SMAsQueue);//use the
Memory area of the SMAsQueue
   }
   createInitSMAMethod.setArguments(new Object[] { thetypeOfMemory
});//assign the arguments of the createInitSMAMethod, which has the
encapsulated method

   if (SMAs_InitialSizes.length > 1) { //initialize a set of SMAs
    sm.executeInArea(createInitSMAMethod);//execute the encapsulated
method to create Initial SMA
   } else if (SMAs_InitialSizes.length == 1) {
   //assign the parameters of the encapsulated method's class
    newAttachedSMA.SMAsQueue = SMAsQueue;//assign the queue
    newAttachedSMA.type = 2;//assign the type
    sm.executeInArea(newAttachedSMA);//execute the encapsulated  method
to create new Attached SMA
   }
  }
 }

 /*///////////////createTask() Method/////////////////////
Function:A method to create a task within the ForkedMemoryModel
Paramters-> tasktype:The type of the task, i.e. RTThread, AEH
            schParams:scheduling parameters of the task
            memParams:memory parameters of the task
            relParams:release parameters of the task
            logic:logic of the task
            taskMemoryType:the scoped memory type for the task
            initMemSz:the initial memory size of the task
            maxMemSz:the maximum memory size of the task
 *//////////////////////////////////////////////////////////
 public Schedulable createTask(Schedulable tasktype,
SchedulingParameters schParams, MemoryParameters memParams,
     ReleaseParameters relParams, Runnable logic, Class taskMemoryType,
long initMemSz, long maxMemSz) {
  //this is uded for the schedulable objects
  //ask the objectallocator to reuse or to create the
  //memArea,
  ScopedMemory initMem = null;
  Schedulable task = null;
  IQueue SMAsQueue = ((IQueue)getNamedPortal().getObject("SMAsQueue"));

  if (taskMemoryType == LTMemory.class) {

   initMem = new LTMemory(initMemSz, maxMemSz);//create the SMA memory
of the task
   SMAsQueue.insert(initMem, 0);//add the SMA to the SMAs Queue
  }

  if (taskMemoryType == VTMemory.class) {
   initMem = new VTMemory(initMemSz, maxMemSz);//create the SMA memory
of the task
   SMAsQueue.insert(initMem, 0);//add the SMA to the SMAs Queue
  }

  if (tasktype instanceof RealtimeThread) { //must be in the CMA
   task = new RealtimeThread(schParams, relParams, memParams, initMem,
null, logic);//create the task itself
  }
  return task;
 }

 //should be createResuableSMA
 /*////////////createReusableStackTask() Method///////////
Function:A method to create a REUSABLE task within the
ForkedMemoryModel
Paramters-> poolName:The name of the pool from which the task is
                    retrieved
            schParams:scheduling parameters of the task
            memParams:memory parameters of the task
            relParams:release parameters of the task
            stackLogic:component that has the logic of the task
                    divided into levels
            nLevels: The number of the stack levels of the
                    task's logic
            taskMemoryType: The scoped memory type of the task
            initSz[]:the initial memory size of the task
            maxSz[]:the maximum memory size of the task
 *////////////////////////////////////////////////////////
 public IHandler createReusableStackTask(String poolName,
SchedulingParameters schParams, ReleaseParameters relParams,
     MemoryParameters memParams, IStackLogic stackLogic, int nLevels,
Class taskMemoryType, Class handlerType,
     int [] initSz, int [] maxSz) {
  IHandler task = null;
  //the reusable task is created inside the encaps-handler
note:we use Encapsulated Handler here that
  if (handlerType == EncapsulatedHandler.class) {
//from HPool not from the allocator
//the question here, what if we have multiple handler types, we need to
have multiple pools and this has to be reflected here????
   IHPool theHandlersPool =
(IHPool)ICOM.getHandlerPool(poolName);//Retrieve the pool of handlers
   IEncapsulatedHandler hdlr =
(IEncapsulatedHandler)theHandlersPool.getFreeHandler();//get a free
handler from the pool
   ReusableRunnableStack hlogic =
(ReusableRunnableStack)getObjectAllocator().getInstance(ReusableRunnabl
eStack.class);//get an instance of the Reusable Runnable Stack
   hlogic.setStackLogic(stackLogic);//Assign the stack logic to be
executed by the retrieved handler
   hlogic.setParameters(taskMemoryType, nLevels, new LTMemory[]
{(LTMemory)getContainerMA(), (LTMemory)getCMA()},
       initSz, maxSz, ICOM);//initialize the parameters of the stack
Logic
   hdlr.setParameters(schParams, relParams, memParams, hlogic, CMA,
taskMemoryType, initSz[0],
       maxSz[0]); //initialize the parameters of the retrieved reusable
handler
   IQueue SMAsQueue =
((IQueue)getNamedPortal().getObject("SMAsQueue"));//retrieve the Queue
of the SMAs
   SMAsQueue.insert(hlogic.getMem(2), 0/*id[can be time]*/);//???
   return hdlr;
  }
  return null;
 }
}
```

A.4 The Reusable Objects Allocator Pattern

```
package RTCOM;
import javax.realtime.*;
/*///////////Object Allocator Pattern/////////////////
Function:This class represents an allocator for reusable objects
*/////////////////////////////////////////////////
public class GeneralObjectAllocatorCls
 implements IObjectAllocator {
 ObjectInventoryCls HeadOfTheObjectInventory; //Object
 ObjectCarrier freeCarriers = null; //The free object carriers
 GetClass getClass = new GetClass();
 LTMemory getClassScope = new LTMemory(104, 1024);
 static final Class ObjectCarrier_class = ObjectCarrier.class;
 static final Class ObjectInventoryCls_class =
ObjectInventoryCls.class;
/*//////////////////ObjectCarrier class///////////////////
Function:This is an inner class of a carrier for an object
*////////////////////////////////////////////////////////
 public static class ObjectCarrier {
  Object theObject = null;   //the object itself
  ObjectCarrier next = null; //the following object in the list
 }
 /*//////////////////ObjectInventoryCls class////////////////
Function:This class an inventory for the objects
 *//////////////////////////////////////////////////////////
 public static class ObjectInventoryCls {
  Class ObjectType = null;//The type of the object
  ObjectCarrier theObjectCarrier = null; //The carrier
  ObjectInventoryCls next = null; //The next object inventory
 }

 /*//////////////////instance() Method///////////////////////
Function:Get an instance of the allocator
Parameters: None
 *////////////////////////////////////////////////////////
 public static GeneralObjectAllocatorCls instance() { //create
allocator in the current memory area
  GeneralObjectAllocatorCls allocator = null;
  try { //can not be used from within a normal thread
   RealtimeThread rtThread = RealtimeThread.currentRealtimeThread();
//retrieve the current thread
   MemoryArea ma = rtThread.getCurrentMemoryArea(); //retrieve the
current memory area
   allocator =
(GeneralObjectAllocatorCls)ma.newInstance(GeneralObjectAllocatorCls.cla
ss); //create the instance
  } catch (IllegalAccessException ie) {
   System.out.println("Illegal Access.............");
  } catch (InstantiationException ie) {
   System.out.println("Illegal Access.............");
  }
  return allocator;
 }

 /*//////////////////instance() Method///////////////////////
 Function: Get an instance of the allocator
 Parameters->  ma: The scoped memory area
 *//////////////////////////////////////////////////////////
 public static GeneralObjectAllocatorCls instance(MemoryArea ma)
     { //create allocator in ma whichcan only be the current memory
area or an outer[lower in the stack] memory area???
  GeneralObjectAllocatorCls allocator = null;
  try { //can not be used from within a normal thread
   allocator =
(GeneralObjectAllocatorCls)ma.newInstance(GeneralObjectAllocatorCls.cla
ss); //instance creation
```

```
  } catch (IllegalAccessException ie) {
   System.out.println("Illegal Access.............");
  } catch (InstantiationException ie) {
   System.out.println("Illegal Access.............");
  }
  return allocator;
 }

/*///////////////////findTypeInventory() Method/////////////
Function:searches for the object inventory of the given class
Parmters-> Objtype: The Object class type
 */ ///////////////////////////////////////////////////////// private
ObjectInventoryCls findTypeInventory(Class Objtype) {
  if (HeadOfTheObjectInventory == null)
   return null; //list is empty
  for (ObjectInventoryCls currentInv = HeadOfTheObjectInventory;
      currentInv != null; currentInv = currentInv.next) //loop within
the list to find the required inventory
  {
   if (currentInv.ObjectType == Objtype)
    return currentInv;
  }
  return null; //not in the list
 }

/*///////////////////getTypeInventory() Method///////////
Function: get an object inventory of the given class
Parameters-> ObjClass: The class type
*/////////////////////////////////////////////////////////
 private ObjectInventoryCls getTypeInventory(Class ObjClass) {
  ObjectInventoryCls reqInventory = findTypeInventory(ObjClass); //get
an existing inventory if found
  if (reqInventory == null) {
   try {
//create a new inventory if no one exists
    reqInventory =
(ObjectInventoryCls)createObject(ObjectInventoryCls_class,
        "Exception while creating a new Type Inventory");
   } catch (RuntimeException re) { }
   reqInventory.ObjectType = ObjClass;
   reqInventory.theObjectCarrier = null;
   reqInventory.next = HeadOfTheObjectInventory;
   HeadOfTheObjectInventory = reqInventory;
  }
  return reqInventory;
 }


 /*///////////////////getTypeInventory() Method/////////////
Function:get an object inventory of the given class
Paramters->  typeCls: The class type
MSG:???
*//////////////////////////////////////////////////////////////
 private Object createObject(Class typeCls, String MSG) throws
RuntimeException {
  Object obj = null;
  try {
   LTMemory ma = (LTMemory)MemoryArea.getMemoryArea(this); //get the
memory area of this object
   obj = ma.newInstance(typeCls);                        //create
instance of the given class in the ma memory area
  } catch (IllegalAccessException iae) {
   throw new RuntimeException(iae);
  } catch (InstantiationException ie) {
   throw new RuntimeException(ie);
  }
  return obj;
 }
 /*//////////////////listFreeObjects() Method/////////////
Function: write a list of the free available objects
Paramter->    cls: The class name
 *//////////////////////////////////////////////////////////
 public void listFreeObjects(Class cls) {
  ObjectInventoryCls TypeInventory = getTypeInventory(cls); //get the
inventoryof the given class
  if (TypeInventory.theObjectCarrier == null)
   System.out.println("||>>NO FREE Object");
  for (ObjectCarrier c = TypeInventory.theObjectCarrier; c != null; c =
c.next) {
   System.out.println("||>>" + c.theObject);
  }
 }
 /*//////////////////recycleCarrier() Method/////////
Function:write back the freed object carrier to the allocator
Paramters->  theFreedCarrier: The freed carrier of the object
 *////////////////////////////////////////////////////////
 public void recycleCarrier(ObjectCarrier theFreedCarrier) {
  theFreedCarrier.next = freeCarriers; //attach the freed carrier
  freeCarriers = theFreedCarrier;      //update
 }

/*////////////////getFreeObjectCarrier() Method/////////////
Function:retrieve the freed object carrier from the allocator
*/////////////////////////////////////////////////////////// public
ObjectCarrier getFreeObjectCarrier() {
  ObjectCarrier FreeObjCarrier = freeCarriers;
  if (FreeObjCarrier == null) {
   FreeObjCarrier = (ObjectCarrier)createObject(ObjectCarrier_class,
"Exception during creating a carrier");
  }
  freeCarriers = FreeObjCarrier.next;
  return FreeObjCarrier;
 }

 /*//////////////////getInstance() Method//////////////////
Function: retrieve a free  object of the given type
Parameters-> ObjType:The teype of the required object
*///////////////////////////////////////////////////
 public Object getInstance(Class ObjType) {
  ObjectInventoryCls TypeInventory = getTypeInventory(ObjType);
//Get the inventory of the given type
  if (TypeInventory.theObjectCarrier == null) {
   return createObject(ObjType, "Exception while creating a new Object
"); //create the required object type
  }
  ObjectCarrier theCarrier = TypeInventory.theObjectCarrier;
//get the current first object carrier
  TypeInventory.theObjectCarrier = theCarrier.next;
//update the inventory's first element
  Object reqObj = theCarrier.theObject;
//get the object from the carrier
  recycleCarrier(theCarrier);
//recycle the freed carrier to the allocator
  return reqObj;
 }

 /*//////////////////GetClass class//////////////////////
 Function: retrieve a class from a given object
 *//////////////////////////////////////////////////////////
 public static class GetClass
  implements Runnable {
  Object obj;            //The object
  Class ret;             //Its type
  public void run() {
   ret = obj.getClass(); //The class of the given object
  }
 }

 /*////////////////////getInstanceLike() Method///////////////
Function:retrieve object inst. of the type of another obj
Paramters-> Object: an existing object of the required type
/////////////////////////////////////////////////////////////
 public Object getInstanceLike(Object Obj) {
  getClass.obj = Obj;
  getClassScope.enter(getClass); //execute the call in a scoped mrmory
to automatically reclaim the run-time created memory
  return getInstance(getClass.ret); //return the instance
 }

 /*////////////////////recycle() Method////////////////////////
Function:write back the freed object to the inventory
Parameters-> recycledObject:The freed object - to be recycled
 *////////////////////////////////////////////////////////////
 public void recycle(Object recycledObject) throws RuntimeException {
  getClass.obj = recycledObject; //prepare the getClass
  getClassScope.enter(getClass); //execute the call in a scoped mrmory
to automatically reclaim the run-time created memory
  ObjectInventoryCls typeInv = findTypeInventory(getClass.ret);
//search for the inventory of the required type
  if (typeInv == null) {
   throw new RuntimeException("Recycling an Object which was not
created by the allocatpr");
  }
  ObjectCarrier carrier = getFreeObjectCarrier(); //retrieve a free
carrier
  carrier.theObject = recycledObject;//att. the obj to the carr.
  carrier.next = typeInv.theObjectCarrier;//update the carrier
  typeInv.theObjectCarrier = carrier;//update the inventory
 }
```

A.5 The Wedge Thread Pattern

```
/*//////////////WEDGE THREAD Pattern//////////////////*/
//This pattern for pinning a single scoped memory area
/*////////////////////////////////////////////////////*/
public static class WedgeThread
 extends RealtimeThread {
 boolean terminate = false; //condition to terminate
 int priority;              // priority of the wedge thread
 MemoryArea commem;         // the common memory area
 MemoryArea wedgedMem;      //the pinned memory area

 /*/////////Constructor/////////////
 Function: This method is used initialize the wedeg thread
 parameters->pri:priority of the wedge thread
                   CommonMemory:the common memory area
 *///////////////////////////////////////////////
 WedgeThread(final int pri, final MemoryArea CommonMemory) {
  super(new PriorityParameters(pri), null, null, CommonMemory, null,
new Runnable() {
   public void run() {//initiate the params of the wedge thread
    WedgeThread.this.terminate = false;
    WedgeThread.this.priority = pri;
    WedgeThread.this.commem = CommonMemory;
    WedgeThread.this.run(); //excecute the wedge thread’s logic
   }
  });
 }

 /*/////////lockMA() Method/////////////
Function: This method is used to pin a certain memory area
 parameters->    ma:the memory area to be pinned
 */ /////////////////////////////////////
 public void lockMA(MemoryArea ma) {
  wedgedMem = ma; //initialize the memory area
  synchronized (this) {
   try {
    notifyAll(); //notify the wedge thread to propagate
   } catch (Exception ex) {
    System.out.println("MA" + ma + "is Free Now");
   }
  }
 }

 /*////////unLockMA() Method////////////
 Function: This method is used to unpin a certain memory area
 parameters: no assigned parameters
 *////////////////////////////////////////
 public void unLockMA() {
  synchronized (wedgedMem) {
   try {
    wedgedMem.notifyAll(); //notify the wedget thread
   } catch (Exception ex) {
    System.out.println("MA" + wedgedMem + "is Free Now");
   }
  }
 }

 /*////////run() Method////////////
Function:This method defines the logic of the wedge thread
parameters:     no assigned parameters
 *//////////////////////////////////////
 public void run() {
  while (!terminate) // the thread until terminated by the user
  {
   synchronized (this) {
    try {
     wait(); //wait in the common memory area until notified to execute
    } catch (Exception ex) {
     System.out.println("MA" + wedgedMem + "is Free Now");
    }
   }
//here the wedge thread is activated to lock a memory area
//the wedge thread runs the following logic
   wedgedMem.enter(new Runnable() { //the Runnable should be made
reusable
    public void run() {
     synchronized (wedgedMem) {
      try {
       //The thread waits in the memory area to pin it
       wedgedMem.wait();
      //The thread here is activated to exit the meory area
      } catch (Exception ex) {
       System.out.println("MA" + wedgedMem + "is Free Now");
      }
     }
    }
   });
  }
 }

 /*////////startNewInstance() Method////////////
Function:This method starts the execution of the wedge thread
parameters->  pri:priority of the wedge thread
               ma:the memory to be pinned
 *///////////////////////
 public static WedgeThread startNewInstance(int pri, MemoryArea ma) {
  WedgeThread w = new WedgeThread(pri, ma); //create the wedeg thread-
could be reusable
  w.start();  //start the wedge thread
  return w; //return wedeg thread reference
 }
 /*////////kill() Method////////////
Function:This method terminates the wedge thread
parameters->no assigned parameters
 */ /////////////////////
 public void kill() {
  terminate = true; //set the termination condition
  synchronized (wedgedMem) {
   try {//activate the wedeg thread to exit the pinned mem. area
    wedgedMem.notifyAll();
   } catch (Exception ex) {
    System.out.println("MA" + wedgedMem + "is Free Now");
   }
  }

  synchronized (this) {
   try {//activate the wedeg thread to exit the common mem. area
    notifyAll();
   //the wedeg thread should terminate now
   } catch (Exception ex) {
    System.out.println("MA" + wedgedMem + "is Free Now");
   }
  }
 }
}
/////////////END OF WEDGE THREAD///////////
```

A.6 The Dual Fork Thread Pattern

```
package RTCOM;
import javax.realtime.*;
       /*//////////////////DualFork Class///////////////////////
Function: Has the logic of the Dual Fork Pattern
////////////////////////////////////////////////////////////*/
public class DualFork
 implements IMemoryModelControler {//A class implements dualFork
Pattern
 IObjectAllocator allocator; // a reference to the allocator object
 int innerForks = 0;
 public ForkThread head1;
 public ForkThread head2;
 public int turn = 2;
 ScopedMemory com_mem;
 public Object updateLock;
 int UpdateRequests = 0;

/*////////getMNPortal() Method/////////////
Function: Retrieves the multi-named portal of a certain memory area
parameters->  mem:the accessed memory area
*/////////////////////////////////////////
 public INamedObjectCollection getMNPortal(ScopedMemory mem) {
INamedObjectCollection memPortal =
(INamedObjectCollection)mem.getPortal();
  if (memPortal == null) {                              //create
the multinamed portal if does not exist
   memPortal = new NamedObjectFastMap();
   mem.setPortal(memPortal);//set the created object as a portal
  }
  return memPortal;
 }

 /*////////Constructor///////////////////
 Function:This method starts the execution of the wedge thread
parameters-> baseMemmem:the common memory area
              ForkedMemories:list of inner scoped memory areas; i.e. it
                  is the memHeldQueue defined in the thesis
 */ //////////////////////////////////
 public DualFork(ScopedMemory baseMem, IQueue ForkedMemories) {
  Queue forkedMemories = (Queue)ForkedMemories; //we use here the class
Queue as a specific linked list implementation
  int pri = PriorityScheduler.instance().getMaxPriority(); //retrieve
the maximum priority
  head1 = new ForkThread(new PriorityParameters(pri / 2), baseMem,
forkedMemories, this,"FirstHead");//create the first forked thread
  head2 = new ForkThread(new PriorityParameters(pri / 2), baseMem,
forkedMemories, this,"SecondHead"); //create the second forked thread
  updateLock = forkedMemories; //set the updatelock as the commom
scoped memory
  com_mem = baseMem; //assign the common memory
common memory
  allocator = GeneralObjectAllocatorCls.instance();//create an instance
of the reusable objects allocator
  LTMemory tmpMem = new LTMemory(1500000, 1500000); //create a scoped
memory???
  INamedObjectCollection memPortal = getMNPortal(com_mem); //retrieve
the portal of the common memory
  memPortal.insertObject("tmpMem", tmpMem); //insert the tmp memory to
the portal
  memPortal.insertObject("HEAD1", head1); //safe a reference to the
first fork thread
  memPortal.insertObject("HEAD2", head2); //safe a reference to the
esecond fork thread
  memPortal.insertObject("allocator", allocator); //save a reference to
the allocator
```

```
  memPortal.insertObject("updateLock", updateLock); //save a reference
to the update lock
  head1.start();//start running the first fork thread
  head2.start();//start running the second fork thread
 }


 /*////////updateFork() Method////////////
Function:This method updates the status of the DualFork
parameters->  no parametera
 *//////////////////////////////////////////
 public synchronized void updateFork() {
  try {
   synchronized (this) {
    while (innerForks < 2) //loop as long as the 2 fork threads are
ready
    {
     try {
      wait(); //wait untill be ready
     } catch (Exception nm) {
      System.out.println("!Exception!" + nm);
     }
    }
   }
  } catch (InterruptedException ie) {
      System.out.println("!Exception!"+ie);
  }
  INamedObjectCollection memPortal = getMNPortal(com_mem); //retrieve
the multi-named portal of the common memory
  Object updateLock = memPortal.getObject("updateLock");   //retrieve
the saved updatelock
  synchronized (updateLock) {
   try {
    while (head1.state == 2 || head2.state == 2 || head1.state == 4 ||
head2.state == 4) {
     synchronized (this) {
      UpdateRequests++; //increment the number of unprocessed requests
     }
     updateLock.wait(); // The External Thread Blocked Waiting to be
able to update the fork
     UpdateRequests--;  //reduce the number of unprocessed requests
    }
   } catch (InterruptedException ie) {
     System.out.println("--Ex--"+ie);
   }
  }
  turn = (turn % 2) + 1; //toggle the turn value, to decide which fork
  if (turn == 1) //the turn of the first fork thread
  {
   head1.state = 2;//set 1st fork thread to the forward propagate state
   head1.startTime = rtclock.getTime(); //retrieve the current time
   synchronized (head1) {
    try {
     head1.notifyAll(); //start the propagation of the 1st fork thread
    } catch (Exception w) { }
   }
  }
  if (turn == 2) //the turn of the second fork thread
  {
   head2.state = 2; //set 2nd fork thread to forward propagation state
synchronized (head2) {
    try {
     head2.notifyAll(); //forward propagate the second fork thread
    } catch (Exception w) {
     System.out.println("—Ex-"+w);
    }
   }
  }
 }

/*////////ForkThread Class////////////////////
Function: It is an implementation of the ForkThread Pattern
             It is used to represent the two head of the fork,
             so it is defined as an inner class
*//////////////////////////////////////////////////
 public static class ForkThread
  extends RealtimeThread {
  public int state; //identifier of the exec. state of the ForkThread
  Queue forkedMemories;// nested SMAs i.e. it is the memHeldQueue
  ScopedMemory commonMemory; //the common memory of the fork
  boolean terminateAll; //cond. of terminating exec. of the dual thread
  boolean running = false;//a flag of the running state
  Object runningLock = new Object();
  public String headName; //Name of the Fork Thread
  public String headNameParams;
  public INamedObjectCollection baseMemPortal; //The MNOP
  public DualFork parentFork;//a ref. to the enclosing dual fork

/*///////////////Constructor/////////////////
Function: This constructor initiates the ForkThread
  Parameters-> pp:The priority parameters of the Fork Thread
                CommonMemory: The common memory
                ForkedMemories:a List of all the inner scoped
                   memory areas
                parent:The Dual Fork containing this ForkThread
               hName:Name of this ForkThread
*////////////////////////////////////////////
  public ForkThread(PriorityParameters pp, final ScopedMemory
CommonMemory, final Queue ForkedMemories,
      final DualFork parent, final String hName) {
   //initialize the realtime threads inherited within this class
   super(pp, null, null, CommonMemory, null, new Runnable() {
    public void run() {
     //Initialization is done within the logic of the ForkThread
     parentFork = parent; //assign the parent DualFork
     terminateAll = false; //reset the termination condition
     headName = hName;    //assign the ForkthreadName
     headNameParams = (hName == "FirstHead") ? "FirstHead-params" :
"SecondHead-params"; //Assign the Parameters NAme???
     commonMemory = CommonMemory; //assign common mem of the pattern
     baseMemPortal = parentFork.getMNPortal(commonMemory); //retrieve
the MNOP portal of the common memory
     forkedMemories = ForkedMemories; //assign the list of the scoped
memory areas
     state = 0; //set the state flag of the ForkThread to the ready
state
     ForkThread.this.runIT();
//execute the runIT(), which holds the logic of this ForkThread
    }
   });
  }

  /*////////notifyForkLocK() Method////////////
  Function: This notifyForkLocK() notifies the parent DualFork
  parameters->  No Parameters
  *////////////////////////////////////////////////
  public synchronized void notifyForkLocK() {
   try {
    if (!running) {
     synchronized (runningLock) { //....waiting for runninglock
      runningLock.wait();
     }
    }
   } catch (InterruptedException ie) { }

   try {
    parentFork.notifyAll();//activates the parent DualForkThread
   } catch (Exception x) {
        System.out.println(">>>>>>> " + x);
   }
  }

/*////////update() Method////////////////////
Function:This updates the status of the ForkThread to the required
state
parameters-> No Parameters
*/////////////////////////////////////////////
  public void update() {
   INamedObjectCollection memPortal =
       parentFork.getMNPortal(commonMemory); //retrieve the portal of
the common memory area
   final QueueItem qhead = (QueueItem)forkedMemories.getHead(); //Get
the Item holding the First Scoped Memory area
   final ScopedMemory firstbranch = (ScopedMemory)(qhead.item());
//Get the first scoped memory area
   IObjectAllocator allocator =
       (IObjectAllocator)baseMemPortal.getObject("allocator");//Get the
reusable objects allocator
   propagate(qhead); //Start the propagation of this fork thread
   ForkThread currHead =
(ForkThread)(RealtimeThread.currentRealtimeThread()); //Get the current
ForkThread reference
   ForkThread otherHead;
   //The if statement >> assigns the other fork thread reference
   if (parentFork.head1 == currHead)
    otherHead = parentFork.head2;
   else
    otherHead = parentFork.head1;
   otherHead.state = 3; //change the state of the other thread
   currHead.state = 1;  //change the state of the current thread to the
propagating state
  }
/*////////propagate() Method///////////////////
Function: This is a recursive function that propagates the Fork Thread
 in the required list of scoped memory areas
parameters-> startingItem: The first item in the list holding the
scoped memory areas that needs to be pinned
  *///////////////////////////////////////////
  private void propagate(QueueItem startingItem) {
   QueueItem item = startingItem; //Get the first item
   ForkThread forkHead =
(ForkThread)RealtimeThread.currentRealtimeThread();//Get the current
fork thread
          INamedObjectCollection memPortal =
parentFork.getMNPortal(commonMemory);//Get the MNOP portal of common
memory of the dualfork
   IObjectAllocator allocator =
       (IObjectAllocator)memPortal.getObject("allocator"); //Get the
reusable object allocator
   if (item.next() == null) {
    Runnable r =
(TailRunnable)allocator.getInstance(TailRunnable.class); //Get Free
Object of the TailRunnable class
    r.run();//execute the logic of the TailRunnable
    //the fork thread here propagated in the last scoped memory area
and have to stop propagation
    return; //exit the recursive function to stop the recusrion
   }//end of if statement
   ScopedMemory branch = (ScopedMemory)(item.item()); //Get the item
holding the required scoped memory area
   TMPBranchRunnable rbranch =
(TMPBranchRunnable)allocator.getInstance(
       TMPBranchRunnable.class); //Get a free instance of the Runnable
to propagate within a scoped memory area
```

```
   rbranch.ITEM = item; //assign the next item that holds the next SMS
to be pinned
   branch.enter(rbranch); //the function calls itself to propagate
recuresively in the next scoped memory areas
  }
/*////////runIT() Method////////////////////
Function: This function holds the logic to be executed by the
ForkThread
parameters-> No Parameters
*/////////////////////////////////////////////
  public void runIT() {
   running = true; //raise the running Flag
   try {
    synchronized (runningLock) {
     runningLock.notifyAll(); //activate any waiting Thread
    }
   } catch (InterruptedException ie) { }
   IObjectAllocator allocator =
       (IObjectAllocator)this.baseMemPortal.getObject("allocator");
//Get the resuable objects allocator
   Runnable r = (ForkRunnable)allocator.getInstance(
       ForkRunnable.class); //Create an instance of the ForkRunnable,
which holds the logic
   r.run();
//execute the Fork logic//?????
  }
/*/////////////////TailRunnable Class//////////////////////
Function:Executes the logic to be executed in the last scoped memory
area to pin it
///////////////////////////////////////////////////////////////*/
  static public class TailRunnable
   implements Runnable {
   public void run() {
    ForkThread forkHead =
(ForkThread)RealtimeThread.currentRealtimeThread();//Get the current
real-time thread
    ScopedMemory com_mem = (ScopedMemory)forkHead.commonMemory; //Get
the common memory of the pattern
    ForkThread otherHead;
    INamedObjectCollection memPortal =
forkHead.parentFork.getMNPortal(com_mem); //Get the portal of the
common memory
    {
     try {//distinguish between the current and the other fork
      if (forkHead.parentFork.head1 == forkHead)
       otherHead = forkHead.parentFork.head2; //set the other fork
      else if (forkHead.parentFork.head2 == forkHead)
       otherHead = forkHead.parentFork.head1;
      forkHead.state = 3; //assign the state of the current fork
      Object forkLock = forkHead;
      while (forkHead.state == 3) {
       synchronized (forkLock) {
        try {
         Object updateLock = memPortal.getObject("updateLock");//Get
              the update lock
         if (otherHead.state == 3) {
          otherHead.state = 4; //forward to the next state
          synchronized (otherHead) {
           otherHead.notifyAll();//activate the other fork
          }
         }
         if (forkHead.parentFork.UpdateRequests != 0) {
          synchronized (this) {
           synchronized (updateLock) {
            try {
             updateLock.notifyAll();
            } catch (InterruptedException i) { }
           }
          }
         }
         forkLock.wait();
         forkHead =
(ForkThread)RealtimeThread.currentRealtimeThread();//Get the current
thread
        } catch (Exception ex) {
          System.out.println("_____ERROR_____" + ex);
        }
       }
      }
     } catch (InterruptedException ie) {
       System.out.println(ie);
     }
    }
   }
  }
  /*////////////////////////TMPBranchRunnable//////////////////
  Function: This class Encapsulates the logic for pinning the scoped
             memory areas, except the last one
 /////////////////////////////////////////////////////////////*/
  static public class TMPBranchRunnable
   implements Runnable {
   public QueueItem ITEM; //Item holding a scoped memory area
   public TMPBranchRunnable() { }
   public void run() {
    ScopedMemory branch = (ScopedMemory)(ITEM.item());
//Get the scoped memory area from the item
    ForkThread forkHead =
(ForkThread)RealtimeThread.currentRealtimeThread(); //Get the current
ForkThread
    ScopedMemory com_mem = forkHead.commonMemory;//set the common
memory
    INamedObjectCollection memPortal = forkHead.baseMemPortal;//set the
MNOP portal
    final QueueItem NEXTITEM = ITEM.next(); //Retrieve the item that
holds the next memory area
    IObjectAllocator allocator =
(IObjectAllocator)forkHead.baseMemPortal.getObject(
        "allocator"); //retrieve a referencxe to the reusable objects
allocator
    BranchRunnable branch2 =
(BranchRunnable)allocator.getInstance(BranchRunnable.class);
    branch2.NEXTITEM = NEXTITEM;
    if (branch != com_mem) {
     INamedObjectCollection nestedmemPortal =
(INamedObjectCollection)branch.getPortal();
     if (nestedmemPortal == null) {
      com_mem.executeInArea(branch2);
     } else {
      Queue nestedMemories =
(Queue)nestedmemPortal.getObject("nestedMemories");
      final QueueItem qhead = (QueueItem)nestedMemories.getHead();
      final ScopedMemory firstbranch = (ScopedMemory)(qhead.item());
      propagate(qhead);
     }
    } else {
    //not implemented
    }
   }
  }

public static class BranchRunnable
 implements Runnable {
 public QueueItem NEXTITEM;
 public void run() {
    ForkThread fth =
(ForkThread)RealtimeThread.currentRealtimeThread();
```

```
    ScopedMemory com_mem = fth.commonMemory;
    fth.propagate(NEXTITEM);
   }
  }

  /*////////////////ForkRunnable Class//////////////////////
  Function: This class ???
  ///////////////////////////////////////////////////////////*/
  static public class ForkRunnable
   implements Runnable {
   public ForkRunnable() { }
   public void run() {
    ForkThread forkHead =
(ForkThread)RealtimeThread.currentRealtimeThread(); //Get the current
ForkThread
    ScopedMemory com_mem = forkHead.commonMemory;//assign the common
memory
    boolean terminateAll = forkHead.terminateAll; //???
    INamedObjectCollection memPortal =
        forkHead.parentFork.getMNPortal(com_mem); //Get the MNOP portal
of the common memory
    Object updateLock = memPortal.getObject("updateLock"); //Get the
<updateLock> form tne MNOP portal
    try {
     synchronized (forkHead.parentFork) {
      if (forkHead.parentFork.innerForks == 0) {
       //This is the first ForkThread
       forkHead.parentFork.innerForks++; //increment the number of
running inner Forked Thread
       try {
        forkHead.parentFork.wait();//wait for notification of the
existence of the two running ForkThreads in the Dualfork
       } catch (InterruptedException ss) { }
      }
      if (forkHead.parentFork.innerForks == 1) {
      //This is the second ForkThread
       forkHead.parentFork.innerForks++; //increment the number of
running inner Forked Thread
       try {
        forkHead.parentFork.notifyAll(); //notification that the two
ForkThread required for this Dualfork ara now avaialable
       } catch (InterruptedException ss) {
        System.out.println("......errr......." + ss);
       }
      }
     }
    } catch (InterruptedException ie) {
     System.out.println("xxxxxxxExceptionxxxxxxxxxx\n" + ie);
    } catch (Exception ie) {
     System.out.println("xxxxxxxExceptionxxxxxxxxxx\n" + ie);
    }
    while (!terminateAll) {
     //Now the logic DualForkPattern works as long as the user has not
terminated it
     synchronized (forkHead) {
      try {
       forkHead.state = 1;          //set the propagation state of the
currently executing forkThread
       while (forkHead.state == 1 && forkHead.parentFork.UpdateRequests
== 0) {
        forkHead.wait();//wait for requests for update
       }
       synchronized (updateLock) { //only one Fork Thread can propagate
on a time
        try {
         updateLock.notifyAll(); //notify any object waiting for the
update command
        } catch (InterruptedException i) {
          System.out.println(">>>Exception>>>" + i);
        }
       }
      } catch (Exception ex) {
       System.out.println(">>>>Exception:>>>" + ex);
      }
     }
     forkHead.update(); //Now execute the propagation within th inner
scoped memory areas
    }   //end of while loop
   }
  }
/*///////////////NotifyForkLockRunnable/////////////////////
Function:notify the lock
/////////////////////////////////////////////////////////////*/
  static class NotifyForkLockRunnable
   implements Runnable {
   public void run() {//called by any external thread using this
pattern
    ForkThread forkHead =
(ForkThread)RealtimeThread.currentRealtimeThread(); //Get the current
ForkThread
    ScopedMemory com_mem = (ScopedMemory)forkHead.commonMemory;
//Get the common memory
    INamedObjectCollection memPortal =
        forkHead.parentFork.getMNPortal(com_mem); //Get the MNOP portal
of the common emory
    synchronized (forkHead.parentFork) {
     try {
      forkHead.parentFork.notifyAll(); //notify the parent Dual Fork
     } catch (InterruptedException ie) {
         System.out.println(ie);
      }
     }
    }
   }
  }
 }
```

A.7 The Handlers Pool Pattern

```
package RTCOM;
import javax.realtime.*;
import java.rtnio.channels.*;
/*//////////////////////HandlersPool Class///////////
Function: This class is used to represent a pool of
       Executors/handlers
////////////////////////////////////////////////*/
public class HandlersPool
 implements IHPool {
 public int maxHandlers;//The maximum number of handler within the pool
 public int freeHandlers = maxHandlers; //the number of free handlers
initially is the size of the pool
 IQueue HPool;                          //
 /*////////////////getSize() Method/////////////
Function:retrieve the size of the pool
     Parameters:No parameters
 *//////////////////////////////////////////////
 public int getSize() {
  return maxHandlers;
 }

 /*////////////////HandlersPool() Method/////////////
Function: retrieve the size of the pool
Parameters:
         HandlerType: the type of the handlers to be used
         Size: the total size of the pool
         PoolMA:The memory area in which the pool would be
             created
         HMemType: The memory type of the handlers
         initHMemSize:initial Size of the memory
         maxHMemSize: maximum Size of the memory
 *//////////////////////////////////////////////
 public HandlersPool(Class HandlerType, int Size, MemoryArea PoolMA,
Class HMemType, long initHMemSize,
     long maxHMemSize) {
  //the Pool should be in the cma
  try {
   HPool = (IQueue)PoolMA.newInstance(Queue.class); //creation of the
handlers pool
  } catch (Exception ex) {
   System.out.println("Error in hnadler pool");
  }

  maxHandlers = Size; //initialize the size
  //The creation of all the required handlers
  //assuming fixed size memory
  for (int i = 0; i < Size; i++) {
   IEncapsulatedHandler hdlr;

   try {
    hdlr = (IEncapsulatedHandler)PoolMA.newInstance(HandlerType); //The
creation of a handler
   } catch (Exception ex) {
    System.out.println("Error in hnadler pool");
   }

//parameters should be set by the user, but these are default
parameters that can
//we should set here the non changable parameters, and outside thuser
set the changable ones
   hdlr.setParameters(null, null, null, null, PoolMA, HMemType,
initHMemSize,
       maxHMemSize); //assign the parameters of the handler
   HPool.append(hdlr); //add the handler to the pool
  }
 }

 /*//////////getFreeHandler() Method////////////
Function:retrieve a free handler from the pool
Parameters: No Parameters
 *////////////////////////////////////////////
 public IHandler getFreeHandler() {
  QueueItem qHead = (QueueItem)HPool.getHead();                    //get
the head of the pool
  IEncapsulatedHandler hdlr = (IEncapsulatedHandler)qHead.item(); //get
the handler from this element
  HPool.remove(qHead);
//remove this element form the pool
  freeHandlers--;
//decrement the count
  return hdlr;
 }

 /*/////////////getFreeHandler() Method/////////
Function: retrieve a free handler from the pool
Parameters:reqMemSize: The required memory size ofthe
              Handler
 *///////////////////////////////////////////
 public IHandler getFreeHandler(long reqMemSize) {
  IEncapsulatedHandler hdlr = null;

  synchronized (HPool) {
   QueueItem Tail = (QueueItem)HPool.getTail();//get the last item

   while (true) {
    QueueItem qi = (QueueItem)HPool.getHead();//get the first item

    try {
     while (qi == null)
      HPool.wait(); //wait if no free handlers are available
    } catch (Exception ex) {
     System.out.println("Exception occured");
    }
    hdlr = (IEncapsulatedHandler)qi.item(); //get the handler

    if (hdlr.getMaxSz() >= reqMemSize) //check if the memory size  is
acceptable
    {
     HPool.remove(qi); //remove the item form the pool
     break;//we found the handler, so finish the loop
    }

    if (qi == Tail)
     break;
    HPool.remove(qi); //
    HPool.append(qi);
   }
  }
  freeHandlers--; //decrement the free handlers count
  return hdlr;
 }

 /*///////////appendHandler() Method/////////////
Function: add a handler to the pool
Parameters-> ehdlr: a handler to be appended
 *////////////////////////////////////////////
 public void appendHandler(IHandler ehdlr) {
  synchronized (HPool) {
   HPool.append(ehdlr); //add the handler
   freeHandlers++;      //increment the free handlers count

   try {
    HPool.notifyAll();  //notify that a free handler is added
    } catch (Exception ex) {
     System.out.println("Exception occured");
    }
   }
  return;
 }
}
```

A.8 The Reusable Runnable Stack Pattern

```
package RTCOM;
import javax.realtime.*;
/*///////////RunnableStack Pattern/////////////////////
Function:Represents pattern for a memory stack
/////////////////////////////////*/
public class ReusableRunnableStack
 implements Runnable
    { //[useful for nesting]it is reusable if we do keep the creation
of the SMA[1-depth] un the initialization as the memory areas are not
deleted, but it can be reusable if we moved the initialization in the
run method see enhanced version runnableStackVersion 2
 //needs to be created in the container to be able to be started from
it???notsure
 LTMemory [] SMA; //Array holds the SMAs currently created
 int curLevel = 0;//current level in which the thread is running
 int nInitLevels = 0; //Number of inital levels in the stack
 boolean initialized = false; //A flag to indicate if it is initialized
or no
 int nLevels; //Number of levels in the stack
 IStackLogic theStackLogic; //A reference to the Satck Logic running
within this memory stack
 ForkedMemoryModel theForkedMemModel; //A refernece to the Memory model
containing this Memory Stack
 int [] initSzs; //Array of the initial sizes of the inital scoped
memory areas
 int [] maxSzs; //Array of the MAX sizes of the inital scoped memory
areas
 IComponent theParentComponent; //a reference to the component holding
this memory stack

 /*//////////////getMem() Method///////////////
Function: This method is used to get one of the initial memory models
Parametrs->  memLevel:The index of the required initial level
 ////////////////////////////////////////////*/
 public LTMemory getMem(int memLevel) {
  return SMA[memLevel];
 }

 /*//////////////Constructor//////////////
Function: This method called when instance is created
Parametrs->  No Paramnters
 ////////////////////////////////////////////*/
 public ReusableRunnableStack() { }

 /*//////////////setStackLogic() Method///////////////
Function:This method is used assign the Stack Logic, which
is to be executed within this stack scoped memory
Parametrs-> stackLogic:ref. to the required Stack Logic comp.
 ////////////////////////////////////////////*/
 public void setStackLogic(IStackLogic stackLogic) {
  theStackLogic = stackLogic;
 }

 /*//////////////setParameters() Method///////////////
Function: This method is used to assign the Parameters of this Memory
Stack
Parametrs-> MemType: The Memory type of the scoped memories within this
memry stack
             nlevels: Total number of initial levels
             initMems[]: An array for inital scoped mem areas
             InitSzs[] :  Initial Sizes of the Initial Memories
            MaxSzs[]  :   Max Sizes of the Initial Memories
            ParentComponent: A reference to the Component
                          containing this memory stack
 ////////////////////////////////////////////*/
 public void setParameters(Class MemType, int nlevels, LTMemory
initMems [], final int [] InitSzs, final int [] MaxSzs,
     IComponent ParentComponent) {
  theParentComponent = ParentComponent; // the parent component
  initialized = false; //flag for initialization
  curLevel = 0; //initialize the curLevel flag to the first level???
  initSzs = InitSzs; //initialize the initSzs of the class
  maxSzs = MaxSzs;   //initialize the maxSzs of the class
  //Here we consider the LTMemory class only, other scoped memory area
types can be implemnted the same way
  if (MemType == LTMemory.class) {
   nLevels = nlevels; //Initialize the total numbers of the initial
levels
   SMA = new LTMemory[nLevels]; //create the initialMemories array of
the assigned size
   if (initMems != null) {
    initialized = true;  //set the initialization flag
    nInitLevels = initMems.length; //initialize the total number of
initial levels???
    for (int i = 0; i < nInitLevels; i++) //set all the assigned
initial scoped memory areas
      SMA[i] = initMems[i];
   }
  }
 }

/*//////////////run() Method///////////////
Function:This method runs the Runnable logic of this class
Parametrs: No Parameters
 ///////////////////////////////////////////////*/
 public void run() {
  logic.run();
 }
 //The following logic Parameters is a Runnable that holds the
execution logic of this component
 protected Runnable logic = new Runnable() {
  public void run() {
   synchronized (this) {
    repeatable = false; //reset the repeataion flag(e.g. used for
periodic case) of this component
    runUpward(curLevel/*-1*/, theParentComponent); //runs the upward
logic of the current level
    if (!isbackward)                              //This code runs
only in the forward phase????
    {
     if (!initialized) {
      SMA[0]
          = (LTMemory)
              RealtimeThread.getCurrentMemoryArea(); //initialize the
lowest memory area to be the current scoped memory area
      initialized = true; //set the initialization flag
     }
     LTMemory nextMem;
     if (++curLevel < nLevels) //check if there are more initial scoped
memories
     {
      if (curLevel >= nInitLevels) {
       if (curLevel == 2)
        nextMem = new LTMemory(initSzs[curLevel - nInitLevels] / 40,
maxSzs[curLevel - nInitLevels]
           / 400);
      }
      if (curLevel == nInitLevels) { //the SMA's mem above the cma is
created so, no it to be added to the list
       INamedObjectCollection namedPortal =
           (INamedObjectCollection)((ScopedMemory)
RealtimeThread.getCurrentMemoryArea()).getPortal(); //retrieve the
portal of the current memory area
       if (namedPortal == null) {
```

```
((ScopedMemory)RealtimeThread.getCurrentMemoryArea()).setPortal(namedPo
rtal); //assign the new named portal
        }
       }
      }
     if (curLevel < nInitLevels)  //runs the CMA only as it already
starts in the contma
      {
       SMA[curLevel].enter(logic); //runs the logic of the attached
logic component
      } else {
       isbackward = true;//here, we finished the forward path, so set
the back ward flag???
        {
        nextMem.enter(logic); //runs the logic in ths nextMem
        }
      }
     } //if(!isbackward)
     runDownward(curLevel--/*-1*/, theParentComponent); //execute the
Downward logic of the logic component
     if (curLevel == -1)// condition for the periodic case only
     {
      repeatable = true; //set the flag
      curLevel = 0;      //reset to the initial level
     }
    }
   }
  };//end Runnable
 protected boolean isbackward = false; //a flag for direction of
propagation
 protected boolean repeatable = true;  //a flag for the periodic
execution
/*//////////////setRepeatable() Method///////////////
Function:This method enable/disable the periodic/repeatable
             execution of this class
Parametrs->  b: a boolean to specify the periodic/repeatable case
 ///////////////////////////////////////////////*/
 public void setRepeatable(boolean b) {
  repeatable = b;
 }
 /*//////////////runUpward() Method///////////////
Function:This method executes the corresponding logic defined in
         the stack logic component each time one of the scoped
         memories is entered
Parametrs->  curLevel: Specifies the index of the current scoped
                       memory area
             theParentComponent: Specifies the parent component
                                 containing this stack memory
 ///////////////////////////////////////////////*/
 public void runUpward(int curLevel, IComponent theParentComponent)
     { //can we get the stack logic from the portal of the mem area
  theStackLogic.runUpWard(curLevel, theParentComponent); //calls the
corresponding method from the logic component
 }
 /*//////////////runDownward() Method///////////////
Function: This method executes the corresponding logic defined in the
stack logic component each time one of the scoped memories is exited
Parametrs-> curLevel: Specifies the index of the current scoped
                      memory area
            theParentComponent: Specifies the parent component
                                containing this stack memory
 ///////////////////////////////////////////////*/
 public void runDownward(int curLevel, IComponent theParentComponent) {
  theStackLogic.runDownWard(curLevel, theParentComponent); //calls the
corresponding method from the logic component
 }
}
```

A.9 The Communicator's Events Handling Loop [NB mode]

```
/*//////////////////////pollNB Class////////////
Function: This methid is used to process monitor and process the events
in the communicator component in the non-blocking mode
///////////////////////////////////////////////////*/

public void pollNB() {
  try {
   for (;;) {
    SelectionKey clientkey;
    if (key == serverkey && key.isAcceptable()) {// case: accept event
     final SocketChannel clientCh = theServerChannel.accept(); //accept
the connection
     isCallActive = true; //set the flag
     clientCh.configureBlocking(false); //set the non blocking mode
     try {
      clientkey = clientCh.register(selector,
          SelectionKey.OP_READ); //the created client cahnnel is
registered to wait for requests here
     } catch (Exception ex) {
      System.out.println(" Can not register client channel");
     }
     IStackLogic logic = (IStackLogic)allocator.getInstance(
         AcceptLogic.class); //Get a free instance of the Accepror
logic using the allocator
     ((AcceptLogic)logic).setSelectionKey(clientkey); //pass the
selection key to the logic
     ((AcceptLogic)logic).setState(1/*state*/); //pass the accept
connection state to the logic
     clientkey.attach(logic); //*** in the non-blocking mode, we attach
the logic instead of a handler as it was in the blocking case
     IHandler acceptHdlr =
         fireNextFreeAcceptHandler(33, clientkey, this, logic); //fire
the handler to execute the logic
    }                //end if isAcceptptable
    else
//client key
    {
     // here the event can be from timer, signal, or network channel
     if (key.channel()instanceof TimerChannel && ComType == 1) {
//in case of a timer channel event
      key.cancel();
//no further processing is made, the key has to be moved away from the
keys set to be removed
      continue;
//we have to jump back to process the next event
     }
//end if

     if (key == SignalKey & key.isReadable()) {
//in case of a read event coming from a signal key
      SignalFDInfo sigInfo = new SignalFDInfo(); //we have to create an
object to read the signal information into it
      theControllerChannel.readSignal(sigInfo); //we have to read the
signal info to remove the event from the channel
      continue;
//we have to jump back to process the next event
     }
//end if
     SocketChannel clientCh;
     if (key != SignalKey) {
//This is a network event[read,connect, or write]
      clientCh = (SocketChannel)key.channel();
//Get the channel of the event
     }
//end if
```

```
     SelectionKey clientkey;
     if (key.isConnectable()) {
//in case of connect event
      clientCh.finishConnect();
//ensure that the ocnnection has been established
      key.interestOps(key.interestOps() | SelectionKey.OP_READ);
//add the read event to be observed on this channel
      key.interestOps(
          key.interestOps() & ~SelectionKey.OP_CONNECT); //do not
observe the connect event of this channel any more
      continue;
//we have to jump back to process the next event
     }
//end if
     if (key.isReadable())//case: a read event on the client channel
     {
      Object attlogic = key.attachment();//Get the attached logic
      SocketChannel ch = (SocketChannel)key.channel(); /get the channel
on which the event occurred
      key.cancel();//remove this key from the keys set to not be
processed any more
      if (ComType == 0) //in the case of the non-blocking server
      {
       AcceptLogic logic = (AcceptLogic)attlogic;
//we use here, for the server, the AcceptLogic
       logic.theChannel = ch;
//logic.setHandler((IEncapsulatedHandler)readHdlr);
       logic.gotoNextState();
//forward the state machine to the next state
       IHandler readHdlr = fireNextFreeReadHandler(33,
key/*clientkey*/, this,
           logic); //fire a free handler to execute the next state in
the logic
      }  //end if
      if (ComType == 1) //in the case of the non-blocking client
      {
       ClientLogic logic = (ClientLogic)attlogic; //we use here the for
the client side, the client logic
       logic.gotoNextState();     ////forward the state machine to the
next state[next write]
       key.attach(logic);       //***The as above, we pass her logic
instead of handler
       IHandler readHdlr = fireNextFreeReadHandler(33,
key/*clientkey*/, this,
           logic); //fire a free handler to execute the next state in
the logic
      }                                 //end if
     }                                  //end if (isReadable)

     if (key.isWritable()) { //in case of a write event on the client
channel
      Object attHandler = key.attachment(); //Get the attachment
      key.cancel();                         //remove this key from the
keys set to not be processed any more

      synchronized (attHandler) {
       attHandler.notifyAll();              //we have to send the
notification, so that the handler can continue execting the next state
      }                                     //end synhronized
     }                                      //end if isWritable
    }                                       //end of else //client key
   }                                        //end of for
  }                                         //end of try
  catch (Exception e) {
   System.out.println("exception" + e);
  } //end catch
 }  //end of poll
```

A.10 The Communicator’s Event Handling Loop [B mode]

```
/*/////////////////////////poll() Method//////////////
Function:Observe and process the registered events on the selector
in the blocking mode
*////////////////////////////////////////////////////
 public void poll() {
  try {
   for (;;) { //important to optimize the code here
    //using immortal is big mistake, unless reusable objects are
presented
     synchronized (CommunicatorCls.this.selector) {
      try {
       if (!CommunicatorCls.this.IsSelectorReady) {
        CommunicatorCls.this.IsSelectorReady = true;//set the ready
flag
        CommunicatorCls.this.selector.notifyAll();//notify any waiting
thread that wants to use the communicator
       }//end if
      } catch (Exception e) { }
     }//end synchronized

    try {//try 1
     int r = 0;
     try {//try 2
      if (CommunicatorCls.this.selector.isOpen())
       System.out.println(">>>>>SERVERWAITING>>>>>>>>>");
      else
       System.out.println(">>>>>>>");

      CommunicatorCls.this.serverkey =
CommunicatorCls.this.theServerChannel.register(CommunicatorCls.this.sel
ector,
          SelectionKey.OP_ACCEPT);//register the server channel for the
accept event
      r = CommunicatorCls.this.selector.select();//observe and wait for
events

registerationQueue.registerAllFCFS(CommunicatorCls.this.selector);//reg
ister all the enqueued channels saved within the registeration queue
      if (CommunicatorCls.this.theControllerChannel.keyFor(selector) ==
null) {//if the controller signal channel was dregistered
       SignalKey =
theControllerChannel.register(CommunicatorCls.this.selector,
SelectionKey.OP_READ);//then rergister it again with the selector fro
the read event
      }//end if
     } catch (IOException e) {
      System.out.println(">error in select");
     } catch (Exception e) {
      System.out.println(">other error in select");
     }//end try 2
    } catch (IOException ioe) {
     System.out.println("Exception..."+ioe);
    }//end try 1

    Set keys = selector.selectedKeys();//Get the updated set of
selected keys
    for (Iterator i = keys.iterator(); i.hasNext(); ) {
     SelectionKey key = (SelectionKey)i.next();//get the next key
     i.remove();//remove the key from the keys set
     /*if_1*/if (!STARTED && ComType == 1) //if it is at the client
side and has not started yet
     {
      if (startType == 1 && key.channel() == CommunicatorCls.this.tch)
//The case the commmunicator proceessing is controlled by a tmer
channel and the timing of this timer channel has come and caused the
```

```
event
      {
       TimerChannel tch = (TimerChannel)key.channel();//Get the timer
channel
       tch.getTimerInfo(tinfo);//get the asociated infromation of the
timer channel and save it in the tinfo object, this has to be called to
clear the event from the channel
       key.cancel();//move the key to the cancelled keys set to be
removed
      //check tch.attachment for security before starting
      } else if (startType == 1 && key.channel()instanceof
SignalChannel) {//The case the channel is a signal channel which
control the start of the event and the signal is fired and caused this
event
       SignalChannel sch = (SignalChannel)key.channel();//Get the
signal channel
       SignalFDInfo sigInfo = new SignalFDInfo(); //create an object
that can gold the signal info
       sch.readSignal(sigInfo);//Read the signal information from the
channel, this has to be called to clear the event from the channel
       key.cancel();//move the key to the cancelled keys set to be
removed
      } else if (startType == 0 && key.channel()instanceof
SignalChannel) {//the case of immediate start and the event is due to
an internal signal channel that is fired now
       SignalChannel sch = (SignalChannel)key.channel();//Get the
signal channel of this key
       SignalFDInfo sigInfo = new SignalFDInfo(); //create an object
capable of holding the signal information
       sch.readSignal(sigInfo);//read the signal infromation and clear
the event
       //we can switch on the value of the attached data, and make the
handling according to the attached object
       if (key == startSignalkey) //in case of the received signal is
the startsignal
       {
        key.cancel();//move the key to the cancelled keys set to be
removed
       } else if (key == pauseSignalkey) //time is set
       {//we can call the pasue here-not impolemented
        key.cancel();
       } else if (key == stopSignalkey) //time is set
       {
        //we can call stop procedure hew -not impolemented
        key.cancel();
       } else if (key == resumeSignalkey) //time is set
       {
                   //we can call resume procedure hew -not impolemented
        key.cancel();
       }
      } else if (theControllerChannel == key.channel()) {
       System.out.println("??????????????");
      } else {       //all events occuring before [start time or start
signal] are canelled
       key.cancel(); //STARTED=true;
       continue;
      }
     }

     STARTED = true; //set the gflag to enable handling of all the
remaining keys

     if (!key.isValid()) {// if this is a hazard
      key.cancel();//then ignore it
      continue;//go back and wait for more events
     }

     if (key == serverkey && key.isAcceptable()) {//in case of accept
```

```
event
      System.out.println("connection made");
      final SocketChannel clientCh = theServerChannel.accept();//accept
the connection and assign a client channel to handel the calling client
      isCallActive = true;//set the flag
      if (clientCh == null) {
       return;
      }
      clientCh.configureBlocking(false);//set the client channel to be
non blocking
      SelectionKey clientkey;

      try { //the created client cahnnel is registered to wait for
requests here
       clientkey = clientCh.register(selector, SelectionKey.OP_READ);
      } catch (Exception ex) {
       System.out.println(" Can not register client channel");
      }
      int defaulthandshakingPriority = PriorityScheduler.MIN_PRIORITY +
30;//specify avalue for the default priority
      IHandler acceptHdlr;
      IStackLogic accepthlogic;

      accepthlogic =
(IStackLogic)theMemModel.getContainerObjectAllocator().getInstance(
           AcceptLogicCls);//Get a free instance of the given acceptor
stack logic from the allocator
      acceptHdlr =
fireNextFreeAcceptHandler(defaulthandshakingPriority, clientkey, this,
accepthlogic);//fire a free acceptor handler with the default priority

accepthlogic.setHandler((IEncapsulatedHandler)acceptHdlr);//assign the
handler of the acceptor logic

((EncapsulatedHandler)acceptHdlr).setSelectionKey(clientkey);//pass the
clientKey to the accept handler to use it for the processing of the
event
      clientkey.attach(acceptHdlr);//attach the accept handler to the
client key
     } else {//in case of events other than the accept event
      if (key.channel()instanceof TimerChannel && ComType == 1) {
       key.cancel();//if this was a timing event(i.e. just timed-
starting signal) and it needs no processing
       continue;//so go back to process more events
      }
      if (key == SignalKey) { //in case of a signal key, i.e. the
Controller has become active
       SignalFDInfo sigInfo = new SignalFDInfo();//then Create an
object to hold the signal information
       theControllerChannel.readSignal(sigInfo);//then read the signal
information
       continue;//so go back to process more events
      }
      //The next code does the processing of the netwok events
      SocketChannel clientCh = (SocketChannel)key.channel();//get the
client channel on which the event occurred
      SelectionKey clientkey;
      if (key.isConnectable()) {//In case of the Connect EVENT
       if (clientCh.isConnectionPending())//if the connection has not
built completely
        clientCh.finishConnect();//then, complete the establishment of
the connection
       ByteBuffer serverBuf = null;
       key.cancel();//The connect event has been processed,
       continue;//So, go back to process more events
      }
      if (key.isReadable()) //from client channel
      {
       Object attHandler = key.attachment();//Get the attached object
with the key
       if (attHandler == null) { //first time to receive bytes
       }
       synchronized (attHandler) {
        key.attachment().notifyAll();//notify the handler to do the
processing, if it is waiting
        key.cancel(); //move the key to the cancelled keys set to be
removed
       }
      }
      if (key.isWritable()) {//in case of write event
       Object attHandler = key.attachment();//Get the handler attached
with the key
       key.cancel();//move the key to the cancelled keys set to be
removed
       synchronized (attHandler) {
        attHandler.notifyAll();//notify the handler to do the
processing, if it is waiting
       }
      }
     }
    }
   } //end of poll loop
  } catch (Exception e) { }
 }
```

A.11 The ClientLogic Class [Emulated Blocking Mode]

```
package RTCOM;
import java.net.*;
import javax.realtime.*;
import java.rtnio.*;
import java.rtnio.channels.*;
import java.rtnio.charset.*;
import java.util.*;
import java.io.*;
import mfr.java.nio.*;
import java.rtnet.InetAddress;

/*/////////////////////ClientLogic Class///////////
Function: This class is used to as the logic component for the Client
in the non-blocking mode
////////////////////////////////////////////////*/
public class ClientLogic
 extends NWHandlerStackLogic {//not written yet

 SocketChannel channel;  //the selectable channel
 ByteBuffer bufout = ByteBuffer.allocateDirect(40);//output buffer
 ByteBuffer bufin = ByteBuffer.allocateDirect(1000); //input buffer
 CommunicatorCls communicator; //the communicator component
 static int PortNo = 2190; //port number
 static java.rtnet.InetSocketAddress remoteAdress; //the server address
 static java.rtnet.InetAddress LocalHost; //the client address

 static {
  try {
   LocalHost = java.rtnet.InetAddress.getLocalHost();//get the client
host
   remoteAdress = new java.rtnet.InetSocketAddress(LocalHost, PortNo);
//create the server address
  } catch (java.rtnet.UnknownHostException ex) { }
 }

 int vvv = 0;//random number
 Clock rc;//realtime clock
 int nn = 0;//number of bytes
 int packetNumber = 0;//the packet order

 public void runUpWard(int curLevel, final IComponent parentComponent)
{
  if (curLevel == 0) { //---=>>>runs in the container memory area
   if (communicator == 0) {
    rc = Clock.getRealtimeClock(); //get the rt clock
    Random aRandom = new Random(); //create a random number generator
    vvv = (int)(aRandom.nextInt(10) + ((rc.getTime().getNanoseconds() /
1000) % 10)) * 1000; //Get a random value
   }

   communicator =
((CommunicatorCls)(parentComponent.getContainer().getComponents().get(
       "Communicator"))); //Get a refrence to the communicator
component from the container
   channel = communicator.makeConnection(remoteAdress, 2190, 25);
//make a connection to the remote machine
  }
//end of if curLevel==0

  if (curLevel == 1) { //runs in the component memory area
   ///////////////SEND PACKET///////////////
   bufout.clear();          //clear the buffer

   for (int i = 0; i < 10; i++) {
    bufout.putInt(i + vvv); //fill the buffer with random numbers
   }
   bufout.flip();//flip the buffer to be ready for the write operation
   channel.write(bufout); //write the bytes from the buffer
  }   //the component memory area

  if (curLevel == 2) {//the temporary scoped memory area
   //The following code displays messages to the user
   //These statements generates hidden obejcts
   System.out.println("The Packet Number " + packetNumber++ + "Has been
written");
   System.out.println("The client started to wait At" + tstart);
  //any other intermediate operation can be done here

  } //end of curlevel==2
 }  //end of runUpward

 public void runDownWard(int curLevel, IComponent parentComponent) {
  if (curLevel == 2) { }

  if (curLevel == 1) { //runs in the component memory area
   ///////////////////////////////
   // receiving the reply
   ////////////////////////////
   bufin.clear(); //clear the output buffer

   try {
    do {
     try {
      communicator.registerationQueue.add(channel,
SelectionKey.OP_READ,
          bufin); //add the channel to the registeration queue to be
registered for the read operation with the bufin as an attachment
      communicator.theControllerChannel.sigQueueToSignalFD(12,
          10); //send the interrupt control signal to enable the
registeration
     } catch (Exception e) {
      System.out.println("Exception....." + e);
     }
     bufin.clear(); //clear the output buffer

     try {
      synchronized (getHandler()) {
       getHandler().wait();//wait for the notification of packet
arrival
      }
      nn += channel.read(bufin); //read from the channel into the
output buffer
     } catch (Exception ex) {
      System.out.println("*----exeption-----*" + ex);
     }  //end catch
    } while (nn < 1000); //read upto 1000 bytes from the output buffer
   } catch (Exception m) {
    System.out.println("---exeption---" + m);
   }


   //decode the values from the buffer
   bufin.flip(); //flip the output buffer

   for (int i = 0; i < 250; i++) {
    try {
     int m = bufin.getInt(); //read the next integer value from the
buffer
    } catch (Exception nb) {
     System.out.println("-Exception-" + nb);
    }
   }
  //Code for Processing the reply packet
  //.............
  //............
  ////////////////////////////////////////////////////////

  }

  if (curLevel == 0) { //runs in the container memory area
   //RESETTING THE VALUES and CLOSING any opened files, etc. if noyt
needed any more
   nn = 0; //reset to initial values to be recyled clean
  }
 }
}
```

A.12 The AcceptLogic Class [Blocking Mode]

```
package RTCOM;
import java.net.*;
import javax.realtime.*;
import java.rtnio.*;
import java.rtnio.channels.*;
import java.rtnio.charset.*;
import java.util.*;
import java.io.*;
import mfr.java.nio.*;
import java.rtnet.InetAddress;

public class AcceptLogic
 implements INWHandlerStackLogic {

 ByteBuffer inbuf = ByteBuffer.allocateDirect(40); //input buffer
 ByteBuffer outBuf = ByteBuffer.allocateDirect(1000); //output buffer

 int nn = 0; //bytes counter
 int n = 0; //bytes counter

 CommunicatorCls communicator;//the communicator component
 int PacketCount = 0; //counter of received packets

 public void runUpWard(int curLevel, IComponent parentComponent) {
  if (curLevel == 0) { //runs in ContMA
   if (communicator == null) {
    communicator =
((CommunicatorCls)(parentComponent.getContainer().getComponents().get(
        "Communicator"))); //get tref to the communicator from the
container
   }
  }//end level 0

  if (curLevel == 1) { //runs in CMA
   try {
    SelectionKey key =
((EncapsulatedHandler)getHandler()).getSelectionKey(); //get the
selection key
    final SocketChannel channel = (SocketChannel)key.channel(); //get
the channel
    communicator.registerationQueue.add(channel, SelectionKey.OP_READ,
        getHandler()); //regiter the channel for the read event (and
attached with the handler) withe the selector of the communicator
    communicator.theControllerChannel.sigQueueToSignalFD(12, 10);
//activates the eselector tp process the registeration
    inbuf.clear();

    do {
     synchronized (getHandler()) {
      getHandler().wait();
     }                          //end synchronized

     try {
      nn += channel.read(inbuf); //read into the buffer
     } catch (Exception w) { }
    } while (nn < 40);
    inbuf.flip();
   } catch (Exception r) {
    Syustem.out.println("Exception--> " + r);
   }
  }                       //end level 1

  if (curLevel == 2) { //runs in temporary scoped memory area

   //the following statement create hidden objects
   System.out.println("The Packet [" + ++PacketCount + "] has been
received");
   Random mm = new Random(); //create randomization object
   //decode and process the received packet [may create objects]
   for (int i = 0; i < 10; i++) {
    int m = inbuf.getInt();
   //.............
   //.................
   }
  }                      //end level 2
 }                       //end runUpWard

 public void runDownWard(int curLevel, IComponent parentComponent) {
  if (curLevel == 2) { } //in temporary scoped memory area

  if (curLevel == 1) {   //in CMA

   //Fill the reply packet
   outBuf.clear();
   for (int i = 0; i < 250; i++) {
    outBuf.putInt(mm.nextInt(250)); //write to the buffer
   }

        //send the packet
   outBuf.flip();
   try {
    channel.write(outBuf); //write to the channel
   } catch (Exception e) {
    System.out.println("*-----Exception----*" + e);
   }
  }

  if (curLevel == 0) { //in ContMA
   //reset the values, recycle any unrequired object, close unwanted
files, channels, ..etc.
   n = 0;
   nn = 0;
  }
 }
}
```

A.13 The ClientLogic Class [Non-Blocking Mode]

```
package RTCOM;
import java.net.*;
import javax.realtime.*;
import java.rtnio.*;
import java.rtnio.channels.*;
import java.rtnio.charset.*;
import java.util.*;
import java.io.*;
import mfr.java.nio.*;
import java.rtnet.InetAddress;

/*///////////////////////////////////////////////////////
Function: This class represents the Reusable
Runnable Logic component of the Client Component
*///////////////////////////////////////////////////////
public class ClientLogic
 implements NWHandlerStackLogicStateMachine { //not written yet
 int MaxState =2; //The logic has two states only, where the first
state is executed by the calling thread, while the second state is
exeuted by the read handler, and this process kis repeated in each
period
 ByteBuffer outbuf = ByteBuffer.allocateDirect(80);   //create output
buffer
 ByteBuffer inbuf = ByteBuffer.allocateDirect(200);   //create input
buffer
 CommunicatorCls communicator; //ref to the communicator comp.
 SocketChannel channel; //the communication channel
 static java.rtnet.InetSocketAddress remoteAdress; //server address
 static int PortNo = 2190;     //port number
 static java.rtnet.InetAddress LocalHost; //local host

 static {                                               //initaites the
addresses objects
  try {
   LocalHost = java.rtnet.InetAddress.getLocalHost();
//getByName/*Address*/("localhos
   remoteAdress = new java.rtnet.InetSocketAddress(LocalHost, PortNo);
  } catch (java.rtnet.UnknownHostException ex) { }
 }

 /*//////////////runUpWard() Method////////////////////
 Function:This method is called each time the client handler enters a
memory level of the client scoped memory stack
 Parameters:  curLevel:The current memory level
              parentComponent:The enclosing business component of this
logic
 *///////////////////////////////////////////////////////
 public void runUpWard(int curLevel, final IComponent parentComponent)
{
  synchronized (this) {//to avoid the concurrent execution of multiple
handlers
   if (curLevel == 0) {//In the container memory area
    if (communicator == null) { //the following code is not within any
state as it will be executed once only
     communicator =
((CommunicatorCls)(parentComponent.getContainer().getComponents().get("
Communicator"))); //get the communicator
     channel = communicator.makeConnection(remoteAdress, 2190, 25);
//make a connection
    }

    if (currentState() == 0) { }//no specific work for the state
    if (currentState() == 1) { }//no specific work for the state
   }
```

```
    if (curLevel == 1) {//Entering the CMA memory level
     if (currentState() == 0) {//the first handler
      writeOutputPacket();//write the packet
      try {
       communicator.registerationQueue.add(getChannel(),
SelectionKey.OP_READ,
          this); //add this schannel with this logic as ana ttachment
to the registseration queue
       communicator.theControllerChannel.openSignalFD(10); //wake up the
selector to process the registeration
      } catch (Exception e) {
       System.out.println("Exception....." + e);
      }
     } //end state 0

     if (currentState() == 1) { }//the other handler
    }  //end level 1

    if (curLevel == 2) {//entering the temporary scoped memory
     if (currentState() == 0) { }//the first handler

     if (currentState() == 1) {//the other handler
      //The following code displays messages to the user
      //These statements generates hidden obejcts
      System.out.println("The Packet Number " + packetNumber++ + "Has
been written");
      System.out.println("The client started to wait At" + tstart);
     //any other intermediate operation can be done here
     } //end state 1
    }  //end level 2
   }   //end runUpward
  }
  /*/////////////runDownWard() Method////////////////////
  Function:This method is called each time the client handler esits a
memory level of the client scoped memory stack
  Parameters:   curLevel:The current memory level
                parentComponent:The enclosing business component of this
logic
  */
  /////////////////////////////////////////////////////
  public void runDownWard(int curLevel, IComponent parentComponent) {
   synchronized (this) {
    if (curLevel == 2) { }

    if (curLevel == 1) {//in the container memory area
     if (currentState() == 0) { }//the 1st handler does no thing

     if (currentState() == 1) {//the second handler runs this
      readInputPacket(); //read the input packet
     }
    }

    if (curLevel == 0) {//in the CMA memory area
     //we may close channel, reset values, ...etc, if not reused in next
periods
     n = 0;
     nn = 0;

     if (currentState() == 0) { }//the first handler does nothing

     if (currentState() == 1) {//the second handler is the last,
      SetState(0); //we reset the state machine, so that it will start
from [state 0] in the next execution cycle
     }
    }
   }
  }
```

```
 /*//////////////writeOutputPacket() Method//////////////////
 Function:write the output packet
 Parameters:None
 *////////////////////////////////////////////////////////
 public void writeOutputPacket() {
  outbuf.clear();

  for (int i = 0; i < 20; i++) {
   outbuf.putInt(i);
  }
  outbuf.flip();

  try {
   channel.write(outbuf); //writing the packet over the channel
  } catch (Exception ex) {
   System.out.println("---Exception----" + ex);
  }
 }

 /*//////////////readInputPacket() Method/////////////////////
 Function:Read the input packet
 Parameters:None
 *//////////////////////////////////////////////////////////
 public void readInputPacket() {
  inbuf.clear();

  try {
   do {
    nn += channel.read(inbuf); //read the packet
   } while (nn < 1000);
  } catch (Exception ex) {
   System.out.println("*-----exeption-----*" + ex);
  }
 }
 /*//////////////DecodeInput Method////////////////////
 Function:Decode the input packet
 Parameters:None
 *//////////////////////////////////////////////////////////
 public void DecodeInput() {
 //decode  the input packet
 //we may ereceive a client propagated priority [p] value here
 //so, we can use it for the current handler, like this
//RealtimeThread.currentRealtimeThread().setSchedulingParameters(new
PriorityParameters(p));
 }
}
```

A.14 The AcceptLogic Class [Non-Blocking Mode]

```
package RTCOM;
import java.net.*;
import javax.realtime.*;
import java.rtnio.*;
import java.rtnio.channels.*;
import java.rtnio.charset.*;
import java.util.*;
import java.io.*;
import mfr.java.nio.*;
import java.rtnet.InetAddress;
/*//////////////////////AcceptLogic Class///////////
Function: This class is used to as the logic component for the Acceptor
handler in the non-blocking mode
/////////////////////////////////////////////////*/
public class AcceptLogic
extends NWHandlerStateMachineStackLogic {

 CommunicatorCls communicator; //The communicator component
 ByteBuffer outBuf = ByteBuffer.allocateDirect(1000); //output buffer
 ByteBuffer inbuf = ByteBuffer.allocateDirect(80);//input buffer
 Random mm = new Random();                              //Randomizer
object
 int nn = 0;
 int MaxState =2;//[state 0->idle but ready],
 [state 1->executed by accept handler], [state 2->executed by read
handler]
 int PacketCount = 0;//number of read packets

    /*///////////////////runUWard() Method////////////////
    Function: This method is called each time the handler enters a
    memory level of the scoped memory stack of the Acceptor handler
    Paramters->      curLevel:The entered level
              parentComponent:The component enclosing this logic
    ///////////////////////////////////////////////////*/
     public void runUpWard(int curLevel, IComponent parentComponent) {
      synchronized (this) {
       if (curLevel == 0) { //runs in ContTMA
        if (communicator == null) {
         communicator =
    ((CommunicatorCls)(parentComponent.getContainer().getComponents().ge
    t("Communicator")));//get the communicator from the container
        }//end if null
       } //end if level=0

       if (curLevel == 1) { //runs in CMA
        if (state == 1) { //executed by accept handler
         readInputPacket();//read incoming packet
        }

        if (state == 2) {
        //do nothing
        }
       }

       if (curLevel == 2) { //runs in the temporary memory
        if (state == 1)      //executed by read handler
        {
         //The following statements may create hidden objects
         System.out.println("The packet number" + PacketCount++ + "has
    been received");
         DecodePacket(); //do some processing on the packet
        }                //end state 1

        if (state == 2) {
        //do nothing
        } //end state=2
       }  ////level 2
      }   //synchronized
     }    //runUpward

    /*///////////////////runUWard() Method////////////////
    Function: This method is called each time the handler exits a memory
    level of the scoped memory stack of the Acceptor handler
    Paramters->      curLevel:The exited level
              parentComponent:The component enclosing this logic
    //////////////////////////////////////////////////*/
     public void runDownWard(int curLevel, IComponent parentComponent) {
      synchronized (this) {
       if (curLevel == 2) { //runs in the temporary memory
       //do nothing
       }

       if (curLevel == 1) { //runs in CMA
        if (state == 1) {
        //do nothing
        }

        if (state == 2) {

         writeOutputPacket(); //Write and send a reply  Packet


         communicator.registerationQueue.add(getChannel(),
    SelectionKey.OP_READ, this);
         communicator.theControllerChannel.sigQueueToSignalFD(12, 10);
        }                    //end state 2
       }                     //end level 1

       if (curLevel == 0) { //runs in ContTMA
        if (state == 1) {
        //do nothing
        }

        if (state == 2) {
         setState(0); //As this is the last point of the execution of
    this state, then reset the state machine
        }
       }
      } //end synchronized
     }
    /*////////// readInputPacket () Method////////////////
    Function: This method is called to read the output packet
    Paramters: None
    *////////////////////////////////////////////////////////////

     public void readInputPacket() {
      inbuf.clear();

      try {
       nn += getChannel().read(inbuf);
      } catch (Exception r) {
       System.out.println("--Exception>>>>>>>>>-" + r);
      }
     }
    /*////////// writeOutputPacket () Method////////////////
    Function: This method is called to write the output packet
    Paramters: None
    *////////////////////////////////////////////////////////////
     public void writeOutputPacket() {
      outBuf.clear();

      for (int i = 0; i < 250; i++) {
       outBuf.putInt(mm.nextInt(250));
      }
      ///////////////reply packet
      outBuf.flip();

      try {
       getChannel().write(outBuf);
      } catch (Exception e) {
       System.out.println("--Exception-->>" + e);
      }
     }

     public void DecodePacket() {
     //<not implemented>
     //decode and process the packet
     //e.g. int p=getPriority(inbuf);
     //RealtimeThread.currentRealtimeThread().setSchedulingParameters(
     //        new PriorityParameters(p));
     }
    }
```

References

A. Wellings, R. K. C., E. D. Jensen, and D. Wells (April 2002). A framework for integrating the Real-Time Specification for Java and Java's remote method invocation. 5th IEEE International Symposium on Object Oriented Real-Time Distributed Computing.

Alexander, C. (1979). A Timeless Way of Building, Oxford University Press.

Alrahmawy, M. and A. Wellings (2007). A model for real time mobility based on the RTSJ. Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems. Vienna, Austria, ACM**:** 155-164.

Alrahmawy, M. and A. Wellings (2009). Design patterns for supporting RTSJ component models. Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems. Madrid, Spain, ACM**:** 11-20.

Alrahmawy, M. and A. Wellings (2009). An RTSJ-based reconfigurable server component, ACM.

Anderson, J. and E. Jensen (2006). The distributed real-time specification for java: Status report.

Appleton B. (2000). "Patterns and Software: Essential Concepts and Terminology." Retrieved June, 2010, from http://www.cmcrossroads.com/bradapp/docs/patterns-intro.html.

Basanta-Val, P., M. Garcia-Valls, et al. (2005). Towards the integration of scoped memory in distributed real-time Java. ISORC 2005.

Basanta-Val, P., M. García-Valls, et al. (2004). "No heap remote objects: Leaving out garbage collection at the server side." Lecture notes in computer science: 359-370.

Benowitz, E. and A. Niessner (2003). "A Patterns Catalog for RTSJ Software Designs." Lecture notes in Computer Science: 497-507.

Beugnard, A., J. Jézéquel, et al. (1999). "Making Components Contract Aware." IEEE Computer **32**(7): 38-45.

Boger, M. (2001). Java in distributed systems: concurrency, distribution, and persistence, John Wiley & Sons, Inc. New York, NY, USA.

Borg, A. and A. Wellings (2003). "A real-time RMI framework for the RTSJ." REPORT-UNIVERSITY OF YORK DEPARTMENT OF COMPUTER SCIENCE YCS.

Broy, M., A. Deimel, et al. (1998). "What characterizes a (software) component?" Software-Concepts & Tools 19(1): 49-56.

Bruneton, E., T. Coupaye, et al. (2006). "The Fractal Component Model and its Support in Java." Software-Practice and Experience 36(11): 1257-1284.

Burns, A. and A. Wellings (2001). Real-time systems and programming languages: Ada 95, real-time Java, and real-time POSIX, Addison Wesley.

Chen, F., T. Repantis, et al. (2005). Coordinated media streaming and transcoding in peer-to-peer systems. 19th International Parallel and Distributed Processing Symposium.

Clark, R., E. Jensen, et al. (1992). An architectural overview of the Alpha real-time distributed kernel. USNIX Workshop on Micro Kernels ans other Kernel Architectures.

Clark, R., E. Jensen, et al. (2002). The Distributed Real-Time Specification for Java: A Status Report. Embedded Systems conference, San Francisco, USA

Cline, M. (1996). "The pros and cons of adopting and applying design patterns in the real world." Communications of the ACM 39(10): 47-49.

Colmenaresy, J., S. Gorappa, et al. (2006). "A Component Framework for Real-time Java." 12th IEEE Real-Time and Embedded Technology and Application Symposium RTAS'06.

Cooling, J. (1991). Software design for real-time systems, CRC Press.

Corsaro, A. and C. Santoro. (2004). Design Patterns for RTSJ Application Development. 2nd JTRES 2004 Workshop, OTM'04 Federated Conferences**:** 394-405.

Corsaro, A. and C. Santoro. (2005). "The Analysis and Evaluation of Design Patterns for Distributed Real-Time Java Software " 16th IEEE International Conference on Emerging Technologies and Factory Automation

Coupaye, T. and J.-B. Stefani (2006). Fractal component-based software engineering. The 2006 Conference on Object-oriented technology: ECOOP 2006. Nantes, France.

Crnkovic, I. (2002). "Component-based software engineering-new challenges in software development." Software Focus **2**(4): 127-133.

D. C. Schmidt (1995). Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching. Pattern Languages of Program Design Reading, MA: Addison-Wesley**:** pp. 529–545.

D. C. Schmidt et al (Aug. 2000). Leader-Followers: A Design Pattern for Efficient Multi-threaded Event Demultiplexing and Dispatching. Proceedings of the 6th Pattern Languages of Programming Conference, Monticello.

de Miguel, M. (2001). Solutions to make Java-RMI time predictable. The 4th IEEE International Symposium on Object Oriented Real-Time Distributed Computing.

Dibble, P. C. (2008). Real-Time Java Platform Programming.

Douglass, B. (2003). *Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems*, Addison-Wesley.

Duran-Limon, H., G. Blair, et al. (2004). "Adaptive resource management in middleware: A survey." IEEE Distributed Systems Online.

Ed Brinksma, G. C., Ivica Crnkovic, Andy Evans; e tal (2003). ROADMAP: Component-based Design and Integration Platforms. Real-Time Operating Systems and Middleware. A. Project IST-2001-34820.

Eric J. Bruno and G. Bollella (2009). Real-Time Java Programming with Java RTS, Prentice Hall.

Etienne, J., J. Cordry, et al. (2006). Applying the CBSE paradigm in the real time specification for Java. 4th international workshop on Java technologies for real-time and embedded systems, Paris, France   ACM.

FP6IPRUNES. (2005). "Survey of Middleware for Networked Embedded Systems, RUNES PROJECT." from http://www.istrunes.org/docs/deliverables/D5_01.pdf.

FOLDOC (1994). "Free On-line Dictionary of Computing (FOLDOC)."   Retrieved June, 2010, from http://foldoc.org/foldoc.cgi?query=distributed+system.

G. Bollella, B. Brosgol, et al. (2006). "The real-time specification for Java (version 1.0.2). Specification JSR-1, Java Community Process."   Retrieved June, 2010, from http://jcp.org/en/jsr/detail?id=1.

G. Sampemane et al (2006). "*HP-RMI : High Performance Java RMI over FM. University of Illinois at Urbana-Champaign*."   Retrieved June, 2010, from <http://www-csag.ucsd.edu/individual/achien/cs491-f97/projects/hprmi.html>.

Gamma, E., R. Helm, et al. (1995). Design patterns: elements of reusable object-oriented software, Addison-wesley Reading, MA.

García, C. (2004). Techniques to include QoS capabilities in Java RMI. EUNICE 10th European summer school and IFIP WG 6.3 Workshop. Tampere (Finland).

Gartner Inc. (2001). "Middleware In Financial Services: an Overview." Gartner Inc., from http://gartner.lib.depaul.edu/gartner_interWeb/research/98900/98904/98904.html.

Gong, L., C. Kulikowski, et al. (1997). An Intelligent Groupware Environment for Real-Time Distributed Medical Collaboration. The 21st symposium on Computer Applications in Medical Care, American Medical Informatics Association.

Grothoff, C. "OVM VM." from http://linux.softpedia.com/get/Programming/Code-Generators/Ovm-1145.shtml.

High Integrity Java Application Project. "HIJA Home Page."   Retrieved June, 2010, from http://www.hija.info/joomla/.

Hu, J., S. Gorappa, et al. (2007). Compadres: a lightweight component middleware framework for composing distributed real-time embedded systems with real-time Java. Middleware 2007, ACM/IFIP/USENIX 8th International Middleware Conference, Newport Beach, CA, USA,, Springer-Verlag New York, Inc.

Huston, S. and D. Schmidt (2001). Desiggn Challenges, Middleware Solutions, and ACE. C++ Network Programming: Mastering complexity with ACE and Patterns. **1**.

Internet 2 Consortium (2010). "Overview of Middleware." Internet 2 Consortium Retrieved June, 2010, from http://middleware.internet2.edu/overview.

Isovic, D., M. Lindgren, et al. (2000). System development with real-time components. ECOOP Workshop - Pervasive Component-Based Systems.

Jammes, F. and H. Smit (2005). "Service-oriented paradigms in industrial automation." IEEE Transactions on Industrial Informatics **1**(1): 62-70.

Java SE API. "Class ByteBuffer."   Retrieved June, 2010, from http://java.sun.com/j2se/1.4.2/docs/api/java/nio/ByteBuffer.html.

Jose A. Ortega-Ruiz, T. Curdt1, et al. "Continuation-based Mobile Agent Migration." Retrieved June, 2010, from http://hacks-galore.org/jao/spasm.pdf.

JSpasm Open Source. "JSpasm State Machine Package."   Retrieved June, 2010, from http://sourceforge.net/projects/jspasm/.

JSR-051. "The JSR 51: New I/O APIs for the JavaTM Platform."   Retrieved June, 2010, from http://www.jcp.org/en/jsr/detail?id=51.

Kanevsky, A., A. Skjellum, et al. (1997). Standardization of a communication middleware for high-performance real-time systems, RTSS97. Realtime Systems Symposium.

Kegel D. (2006). "The c10k problem." from http://www.kegel.com/c10k.html.

Kircher, M. (2001). Lazy Acquisition Pattern. European Pattern Language of Programs conference. Kloster Irsee, Germany.

Kircher, M. and P. Jain (2002). Pooling Pattern,. European Pattern Language of Programs Conference. Kloster Irsee, Germany.

Krishna, A., D. Raman, et al. (2003). "Optimizing the ORB core to enhance Real-time CORBA predictability and performance." Distributed Objects and Applications (DOA).

Krishna, A., D. Schmidt, et al. (2004). Enhancing Real-time CORBA via Real-time Java features. 24th International Conference on Distributed Computing Systems, ICDCS 2004, Tokyo, Japan, Citeseer.

Krishna, A., D. Schmidt, et al. (2003). Real-time CORBA middleware. Middleware for communications. M. Q. New York, Wiley and Sons.

Krishna, A., D. Schmidt, et al. (2003). Towards predictable real-time Java object request brokers. 9th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2003), Washington, DC, USA.

Krishna A.S. and Schmidt D. C. (2004). *Strategies for providing End-to-End Quality of Service in Real-time Java based Real-time CORBA Middleware*.

Krishnaswamy, V., D. Walther, et al. (1998). Efficient implementations of Java remote method invocation (RMI). 4th USENIX Conference on Object-Oriented

Technologies and Systems (COOTS), Santa Fe, New Mexico,, USENIX Association.

Kwon, J., A. Wellings, et al. (2002). Ravenscar-Java: A high integrity profile for real-time Java. In Joint ACM Java Grande/ISCOPE, Seattle, Washington, ACM New York, NY, USA.

Laplante, P. A. (2004). Real-Time Systems Design and Analysis, John Wiley&Sons, IEEE Press.

Lavagno, L., G. Martin, et al. (2003). Fine Grained Patterns for Real-Time Systems. UML for Real: Design of Embedded Real-Time Systems, Springer.

LimeWire Web Site. "LimeWire." Retrieved June, 2010, from http://www.limewire.com/.

Liu, M.-L. L. (2001). On the Power of Abstraction - a Look at the Paradigms and Technologies in Distributed Applications. International Conference of Parallel and Distributed Processing Techniques and Applications (PDPTA'2001), Las Vegas.

Loeser, C., P. Altenbernd, et al. (2002). Distributed video on demand services on peer to peer basis. International workshop on Real-Time LANS in the in the Internet Age, RTLIA.

M. Voelter, e. (2004). Remoting Patterns Foundations of Enterprise, Internet, and Real-time Distributed Object Middlware Wiley and Sons.

Macmillan, I. (1995). "Training: Structuring Distributed Real-time Systems." from http://www.ansa.co.uk/ANSATech/95/Primary/160201.pdf.

Mark A. Miller. (2005). "Understanding H.323—Part I: History and Architecture." Retrieved June, 2010, from http://www.voipplanet.com/backgrounders/article.php/3498736.

Markus Völter, M. Kircher, et al. (2004). Remoting Patterns-Foundations of Enterprise, Internet, and Realtime Distributed Object Middlware Wiley and Sons.

Medvidovic, N. (2003). "The Role of Middleware in Architecture-Based Software Development." International Jouranl of Software Engineering and Knowledege Engineering (IJSEKE) **13**(4): 367-393.

Message-Passing-Interface-Forum (1994). "A message-passing interface standard. ." International Journal of Supercomputer Applications-special issue on MPI **8**(3/4).

Microsoft.com (2002). "Glossary of Networking Terms for Visio IT Professionals." from http://microsoft.com/technet/prodtechnol/visio/visio2002/plan/glossary.mspx.

Möller, A., M. Åkerholm, et al. (2003). Software Component Technologies for Real-Time Systems- An Industrial Prespective-. WiP Session of Real-Time Systems Symposium (RTSS). Cancun, Mexico.

Morse, K., R. Brunton, et al. (2004). An Architecture for Web-Services Based Interest Management in Real Time Distributed Simulation. the Eighth IEEE Distributed Simulation – Real Time Applications Workshop.

MPI-RT (2002). "The Real-time Message Passing Interface (MPI/RT) Forum." Retrieved June, 2010, from http://www.mpirt.org.

Neelamegam, J. (2002). Zero-sided Communication: Challenges in implementing time-based channels using the MPI/RT specification, Mississipi State University. Master.

Newcombe, A. and J. Seraj (2002). Prioritized agent-based hierarchy structure for handling performance metrics data in a telecommunication management system, Google Patents.

Object Management Group (OMG). (2005). "Real-time CORBA Specification." Retrieved June, 2010, from http://www.ois.com/images/stories/ois/real-time%20corba%20specification%2005-01-04%20jan%202005.pdf.

Opengroup, T. "DCE Home Page." Retrieved June, 2010, from http://www.opengroup.org/tech/dce.

P. Wyckoff, S. McLaughry, et al. (1998). "*T Spaces*." IBM Systems Journal Vol 37,Num 3: pp. 454-474.

Palue, S. (2002). "Real-Time Specification of Java (RTSJ)." Retrieved June, 2010, from http://www.developer.com/java/other/article.php/1367671/Real-time-Specification-for-Java-RTSJ.htm.

Pardo-Castellote, G., R. Innovations, et al. (2005). "Omg data distribution service: Real-time publish/subscribe becomes a standard." RTC Magazine 14.

Parker, D., S. Collins, et al. (2004). Building near real-time P-2-P applications with JXTA. Fourth IEEE International Symposium on Cluster Computing and the Grid (CCGrid'04), Chicago, IL, USA.

Pasetti A and P. W (1999). The Component Software Challenge for Real-Time Systems Proceedings of the Real-Time Mission Critical Systems Workshop (held in conjunction with 20th IEEE RTSS). Scottsdale, AZ (USA).

Pizlo, F., J. Fox, et al. (2004). Real-time Java scoped memory: design patterns and semantics. Seventh IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'04), Vienna, Austria

Plášil, F. and M. Stal (1998). "An architectural view of distributed objects and components in CORBA, Java RMI and COM/DCOM." Software-Concepts & Tools 19(1): 14-28.

Plšek, A., F. Loiret, et al. (2008). A Component framework for java-based real-time embedded systems. 9th International Middleware Conference. Leuven, Belgium**:** 124-143.

Plsek, A., P. Merle, et al. (2008). A Real-Time Java Component Model, HAL - CCSD.

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

Ramamritham, K., S. Son, et al. (2004). "Real-time databases and data services." Real-Time Systems 28(2): 179-215.

Raman, K., Y. Zhang, et al. (2005). Patterns and tools for achieving predictability and performance with real time Java. 11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'05), Hong Kong, China

Raman, K., Y. Zhang, et al. (2005). "RTZen: highly predictable, Real-Time Java middleware for distributed and embedded systems." Lecture notes in computer science 3790: 225.

RENAISSANCE Consortium. (1997). "*Client/Server Migration Guidelines For Migration From Centralised To Distributed Systems*." Retrieved June, 2010, from http://www.comp.lancs.ac.uk/computing/research/cseg/projects/renaissance/pdf/D31a.pdf.

Riehle, D. and H. Züllighoven. (1996). "Understanding and Using Patterns in Software Development." Theory and Practice of Object Systems **2**(1): 3-13.

Ruth Tolosa, J. P. M., Miguel A de Miguel M. Teresa Higuera-Toledano, Alejandro Alonso (2003). Container Model Based on RTSJ Services. OTM Workshop

Schantz, R. and D. Schmidt (2001). "Middleware for distributed systems: Evolving the common structure for network-centric applications." Encyclopedia of Software Engineering: 801–813.

Schmidt, D. (2002). "Adaptive and reflective middleware for distributed real-time and embedded systems." Lecture notes in computer science 2491: 282-293.

Schmidt, D. and F. Kuhns (2000). "An overview of the real-time CORBA specification." Computer 33(6): 56-63.

Sharp, D. (1998). Reducing avionics software cost through component based product line development. Software Technology.

Shaw, M. and G. D (1996). Patterns for Software Architectures. Pattern Languages of Program Design, Addison-Wesley.

Siddiqui, F. (1996). "Component Based Software Engineering " Component-Based Software Engineering, IEEE Computer Society Press: 7-15.

Skjellum, A., A. Kanevsky, et al. (2004). "The Real-Time Message Passing Interface Standard (MPI/RT-1.1)." Concurrency and Computation: Practice and Experience 16(S1): Si - S322.

Son, S. and K. Kang (2002). Qos management in web-based real-time data services. 4th International Workshop on Advanced Issues of Electronic Commerce and Web-based Information Systems (WECWIS'02), Newport Beach, CA.

Stankovic, J., S. Son, et al. (1997). Beehive:Global Multimedia Database Support for Dependable, Real-Time Applications. Real-Time Databases and Information Systems, Kluwer Academic Publishers**:** 409-422.

Sun.com (2005). "JXTA Technology Brings the Internet Back to Its Origin." Retrieved June, 2010, from http://java.sun.com/developer/technicalArticles/JXTA/.

Sun Micro Systems Inc. (2004). "JXTA v2.0 Protocols Specification." from https://jxta-spec.dev.java.net/JXTAProtocols.pdf.

Sun Microsystems Inc (2003). *JavaSpaces Principles, Patterns, and Practice*. *JavaSpaces Principles, Patterns, and Practice*, Eric Freeman, Susanne Hupfer, and Ken Arnold,(1999),.

Sun Microsystems Inc. (2003). "*jGuru: Remote Method Invocation (RMI)*." from <http://java.sun.com/developer/onlineTraining/rmi/RMI.html>.

Sun Microsystems Inc. (2004). "*Java Remote Method Invocation Specification*." Revision 1.10, Java™ 2 SDK, Standard Edition, v1.5.0, Retrieved June, 2010, from http://java.sun.com/j2se/1.5/pdf/rmi-spec-1.5.0.pdf.

Sun Microsystems Inc. (2004, June 2010). "*A Jini-based Real-time Telemetry System for F1 Racing Cars*." from http://www.jini.org/files/meetings/eighth/presentations/Cars/Cars.pdf.

Sun Microsystems Inc. (2004). "*JXTA v2.0 Protocols Specification*." from https://jxta-spec.dev.java.net/JXTAProtocols.pdf.

Sun Microsystems Inc. (2005). "*Sun Java Real-Time System, JavaOne Conference in San Francisco*." from http://java.sun.com/javase/technologies/realtime/index.jsp.

Sun Microsystems Inc. "Java Object Serialization Specification." Retrieved June, 2010, from http://java.sun.com/j2se/1.5.0/docs/guide/serialization/spec/serial-title.html.

T. richardso, A. J. W., J.A. Dianes, M.Diaz (2009). Providing Temporal Isolation in the OSGi Framework. JTRES09. Madrid, Spain.

T. Richardson, a. J. W., J.A. dianes, Mi, Diaz (2010). Towards memory Management for Service-Oriented Real-Time Systems. JTRES. Prague, Czech Reppubic.

TABUCHI M, NAKAJIMA K, et al. (2004). "Communication Door: Real-Time Communication Middleware." NEC J Adv Technol 1(3): 176-183.

Tejera, D., A. Alonso, et al. (2007). Predictable serialization in Java. 5th international workshop on Java technologies for real-time and embedded systems, JTRES 2007

Vienna, Austria

Tejera, D., A. Alonso, et al. (2007). RMI-HRT: Remote method invocation-hard real time. the 5th International Workshop on Java Technologies for Real-time and Embedded Systems, JTRES 2007, Vienna, Austria ACM.

Tejera, D., R. Tolosa, et al. (2005). Two alternative rmi models for real-time distributed applications. The Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing ISORC 2005.

The Computing Dictionary (2010). "The Computing dictionary " Retrieved June, 2010, from http://computing-dictionary.thefreedictionary.com.

The JSR-203. "JSR 203: More New I/O APIs for the JavaTM Platform ("NIO.2") " Retrieved June, 2010, from http://www.jcp.org/en/jsr/detail?id=203

The object Management Group (OMG). (2007). "Data Distribution Service for Real-time Systems, v1.2." Retrieved June, 2010, from http://www.omg.org/technology/documents/formal/data_distribution.htm.

Thomas Richardson, A. W. (2010). An Admission Control Protocol for Real-Time OSGi. 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing. Carmona, Spain.

Urbano, P. G. D. A. (2002). Agent Based Approach To Distributed Real-Time Systems Development Third International Symposium on Multi-Agent Systems, Large complex Systems and E-Business.

Voelter, M., M. Kircher, et al. (2004). Remoting Patterns - A Systematic Approach for Design Reuse of Distributed Object Middleware Solutions. IEEE Computing magazine.

W3C (2010). "The World Wide Web Consortium (W3C) Home Page." Retrieved June, 2010, from http://www.w3.org.

Wang, N., K. Parameswaran, et al. (2001). "Towards a Reflective Middleware Framework for QoS-enabled CORBA Component Model Applications." Component Model Applications, IEEE Distributed Systems Online **2**.

Wang, N., D. Schmidt, et al. (2003). "QoS-enabled middleware." Middleware for communications.

Wang, S., S. Rho, et al. (2004). Toward Real-Time Component-based Systems. IEEE International Real-time Systems Symposium (RTSS) Work-In-Progress Session.

Wellings, A. (2004). Concurrent and real-time programming in Java, John Wiley & Sons.

Wellings, A., R. Clark, et al. (2001). "A framework for integrating the Real-Time Specification for Java and Java's remote method invocation." REPORT-UNIVERSITY OF YORK DEPARTMENT OF COMPUTER SCIENCE YCS: 69-72.

Wells, D. (1994). A trusted, scalable, real-time operating system environment. Dual-Use Technologies and Applications Conference Proceedings Utica, **NY**.

Wells, G., A. Chalmers, et al. (2000). "A comparison of Linda implementations in Java." Communicating Process Architectures **58**: 63–75.

الحمد لله رب العالمين

All the praises and thanks be to Allah, the Lord of the worlds, for His favour to me in completing this thesis, praying to Him to accept this as a sincere service for His sake, and for the benefit of mankind.