Java-based Framework for Implementing Soft Real-Time Distributed Applications

José Rodríguez
Departamento de Computación
CINVESTAV-IPN, D. F., México
Email: rodriguez@cs.cinvestav.mx

Dominique Decouchant
LAFMIA UMI 3175 C.N.R.S.-CINVESTAV
CINVESTAV-IPN, D. F., México
Email: Dominique.Decouchant@imag.fr

Sonia Mendoza
Departamento de Computación
CINVESTAV-IPN, D. F., México
Email: smendoza@cs.cinvestav.mx

Christian Mejía Escobar
Departamento de Computación
CINVESTAV-IPN, D. F., México
Email: chrismej@yahoo.com

***Abstract*—Nowadays, due to the development of various research domains such as groupware, multimedia, and WWW among others, distributed applications require more real-time capabilities to efficiently transfer monomedia and multimedia information between cooperating sites. In last years, Java became unavoidable for the development of distributed applications. Although the Real-Time Specification for Java (RTSJ) propose real-time supports, it only remains a suited solution for specific monoprocesor real-time operating systems. Thus, providing dedicated mechanisms to support Java real-time distributed applications on standard operating systems appears as a main challenge.**

In this work, we propose a Java-based framework for designing/implementing soft real-time distributed applications. This framework focuses on providing central advantages: first, it is "flexible" because it makes possible to define several scenarios to implement an application. It is adaptable because it is possible to adapt the class structure to satisfy the application requirements. Finally, it defines a dynamic infrastructure that allows clients and servers to exchange components at runtime.

***Index Terms*—Distributed system framework, middleware, soft real-time, multi-threaded implementation.**

I. Introduction

One of the main problems to cope with when implementing a distributed system is that concerning the platform heterogeneity (different operating systems and hardware architectures). The middleware infrastructures have been specially designed to solve the heterogeneity problem, allowing the execution of the same application among different computer machines based on different architectures. Moreover a same distributed application can be perceived as a control flow that combines distributed modules eventually produced using different programming languages.

Among the most known general purpose middlewares we have the Common Object Request Broker Architecture [6] (CORBA) and the Java Remote Method Invocation [5] (Java-RMI), both are Object Oriented.

However Java-RMI does not support the development of real-time applications. In the other side CORBA supports several languages and services (among them the extension of CORBA-Real-Time [10]) but CORBA is a more complicated technology and is harder to learn.

One of the main characteristics of a Real-Time System (RTS) is the predictability, i.e. the capability to know and to guarantee response times for events. A real-time system must react to stimulus from the environment; the response could be composed of a single or several tasks. RTSs are classified in soft and hard Real-Time Systems [4] in accordance with the consequences that can be produced in the lost of deadlines.

Distributed systems are becoming more popular and distributed RTSs are used now in industry, military, and telecommunication sectors [8]. In this paper we propose a framework to facilitate the implementation of soft real-time distributed applications.

Given that Java RMI was not designed to implement real-time applications, we are interested in implementing mechanisms to allow the propagation of scheduling parameters from a Java Real-Time application to the components in the distributed system. The mechanisms we are proposing are based in a RTSJ implementation that we can find in Internet and some other additional free tools.

This paper is organized as follows: After presenting some related works in section 2, we analyze in section 3 the java tools we have used to implement our framework. In section 4 we describe the architecture of our proposition, in section 5 we highlight the advantages of our tool and finally in section 6 we present our conclusions.

II. Distributed Real-Time Specification for Java and Java Standard tools

In order to provide suited functions and mechanisms to support distributed Java real-time applications, the Java community is currently working on the Distributed Real-Time Specification for Java (DRTSJ) [7] whose goal is to create a Java platform to implement real-time distributed systems. Wellings [13] presents the mechanisms to integrate and extend the RTSJ capabilities with RMI in order to define the basic mechanisms for the DRTSJ. Borg [2] describes a framework

1550-4069/08 $25.00 © 2008 IEEE
DOI 10.1109/ENC.2008.38

IEEE computer society

to support predictability in the remote method invocation. This solution has been taken as the first step in the definition of the DRTS, although the integration of RTJS with the communication mechanisms still remains as an unsolved problematic point [2]. An additional objective of this research in our work is the use and integration of free Java tools that could help us in the propagation of scheduling parameters over the nodes in a distributed system.

Communication Programming: API Sockets

The standard Java runtime provides a very effective support for implementing network communications using objects that represent network connections and data flows [11]. Using the Java API sockets, it is then possible to effectively implement the communication between several JVM-RT instances.

Remote Method Invocation: Java RMI

Java RMI allows implementing object-oriented distributed applications, where objects are located on several computers distributed over the network, and the inter-communications take place via remote method invocations. Although Java RMI does not support real-time applications, we may use it as the basic communication mechanism for client, server and remote interfaces.

Concurrent programming: Multi-Thread programs

Requiring to start and handle several activities in a concurrent way, we used multi-threaded programming [9]. Java offers an API that allows implementing concurrent applications in an easy way. By using the Java threads we represent the intrinsic concurrence of real-time systems. Moreover, we have implemented the server using Java threads in order to have more reliable servers.

Dynamic Load and Reflectivity

Dynamic load and reflectivity are two capabilities of Java which allow us to get information about some classes and to load *bytecodes* in a dynamic way. These two Java main characteristics allow to provide our framework with a dynamic capability. By this mean, the client can send the components needed by the server to supply a service.

Real-Time Specification for Java: RTSJ

The RTSJ [3] constitutes a first representative effort to allow the design and the implementation of Java real-time applications through: 1) a real-time Application Programming Interface (API) i.e. the Package class `javax.realtime` and 2) a real-time Java Virtual Machine (JVM-RT). Both components have been used as the platform for the implementation and test of soft real-time applications [1] and hard real-time applications. However the JVM-RT has been designed to work in a stand-alone way. In spite of this limitation we have taken it as the basic tool for our work.

III. A Real-time Framework Architecture

We have implemented several tests to evaluate some of the most known implementations of RTSJ. These tests have been developed in a Linux-PC. The final result has been to select TimeSys as the tool to implement our framework. TimeSys has been developed using Java 2 Micro Edition (J2ME), an optimized platform that supports the execution of real-time and non real-time Java code [12].

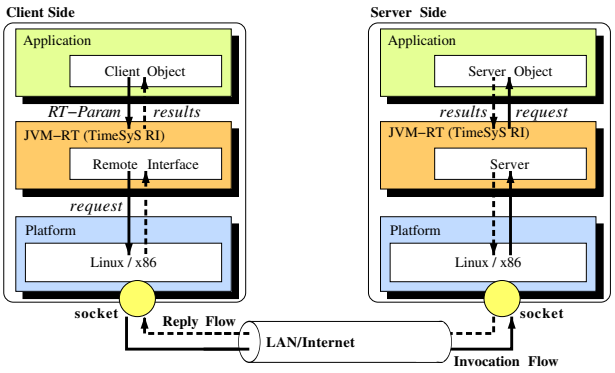

Fig. 1. Components of the Proposed Architecture

Most of the distributed systems are based on the client-server architecture. We have used this architecture to identify the components of our proposition. In Fig. 1 we present the architecture that we have defined to identify the relation between the components of our framework.

In this architecture we have defined the roles of components: we have server and client systems, however roles can be interchanged.

However, every cooperating site has the same components: first the operative platform (composed by hardware an operating system) responsible of establishing the communication between the network and the Java virtual machine (extended with the real-time mechanisms of the RTJS). Second the real-time platform composed of the JVM extended with the mechanisms of the RTJS and the developed classes that can help us to send and get parameters and data for applications, this component is responsible of transforming standard data types into RTSJ real-time parameters.

Thus, the distributed applications developed with our framework are composed of client and server applications which have been developed and executed using TimeSys JVM-RT. A client application could be composed of one or several objects; each one of them could need a remote interface in order to be able to call the methods of objects residing in the server.

However in our case, invoking a remote method implies that the method must be executed under a specific context where scheduling and time parameters must be respected. In our proposed architecture we have replaced the Java-RMI with a JVM-RT using sockets for communication.

Client and server applications are executed in separate computers, so communication is established through sockets. In every method invocation we need to send the real-time parameters, giving the needed support to run this type of applications. To satisfy this requirement, every single node in the distributed system must have an instance of our architecture.

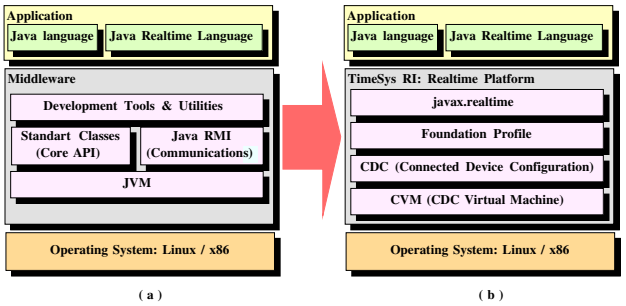

Fig. 2. From a Standard Java Distributed System Towards Our Proposition

A. Towards a Effective Real-time Java Platform

In a conventional middleware JVM architecture, we have that the JVM is responsible of establishing communication through the Java RMI, interpreting and transforming data parameters into the right format for the site, giving some development tools, and supplying standard classes i.e. the Application Programming Interface (API). Giving that Java RMI does not support real-time capabilities and that it is no possible to extend TimeSys RI with the `java.rmi` package. We are proposing, for developing soft real-time distributed applications, to replace the JAVA-RMI for the TimeSys RI RTJS implementation establishing communication between cooperating sites through API sockets of standard Java. TimeSys RI is a development platform based on the Java 2 Micro Edition Connected Device Configuration (J2ME) that does not include the RMI library, i.e. it works only in a stand alone way. The architecture that we are proposing is showed in Fig. 2, and it is composed of:

- CDC Virtual Machine (CVM): this is an optimized version of the JVM, supporting all the characteristics and functionalities of the standard JVM, supporting the CDC and Foundation Profile APIs.
- Connected Device Configuration (CDC): this is an API from the Java 2 Standard Edition (J2SE).
- Foundation Profile: is a set of APIs for the support of network communication and input/output functionalities.
- `javax.realtime` this the real-time API specified in the RTJS.

Every node in the distributed real-time application must to implement this architecture, cooperation is achieved through TCP/IP. As mentioned above, communication is established through sockets for communicating real-time JVMs without RMI, and allowing to send and receive real-time parameters.

B. Class Diagrams

The components in the architecture shown above are defined as Java classes in order to build our framework: These classes constitute a library that works in a closed way with the Java API and the RTSJ (see Fig. 2). This library has the following responsibilities: 1) to handle the concurrent communications in the network, 2) to satisfy the real-time requirements. This library is composed of RTJS and standard Java classes to give the distributed application classes.

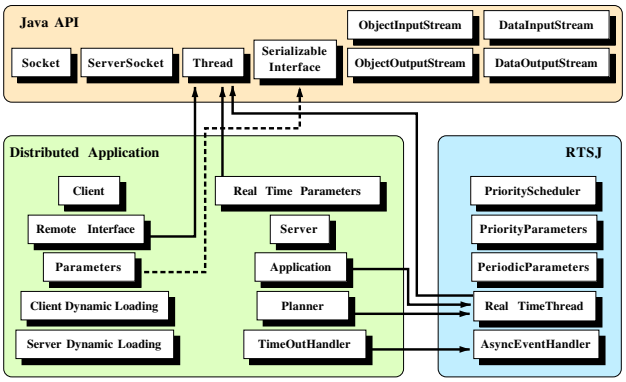

Fig. 3. Classes diagram of our framework

A distributed application, in addition to the basic classes (`Client`, `Server`, `RemoteInterface`), has the `Parameters` class responsible for encapsulating the scheduling and time parameters. These parameters must be serialized in order to sent them through the network, it means that the `Parameters` class implements the Java `Serializable` interface.

The class `RemoteInterface` is a mediator between the invocation of the remote method and the transmission of parameters through the sockets. This class inherits from the `Thread` class, i.e. `RemoteInterface` is treated like a thread with independent execution and where several connections can be managed in a simultaneous way.

The server uses class `RealTimeParameters` to get the parameters and to transform the primitive data types into real-time data types of the RTSJ. Those parameters are the scheduling and time restrictions that must be respected by the `Application` class. This class inherits from the `RealTimeThread` of the RTSJ in order to define a thread of execution with the associated real-time parameters. In this way the original invocation carried out by the client is executed by the server with a thread satisfying the client's parameters.

Additional Classes

To deal with the lost deadlines we have implemented the class `DeadlineHandler`. This class has been developed based on the example from [14] and inherits from RTSJ's `AsynEventHandler` class in order to have an asynchronous event handler.

On the other hand, the real-time threads of a distributed application must be executed in accordance with a scheduling policy. The RTSJ has a priority-based scheduler. In order to have a more flexible scheduler, we have implemented an additional scheduler that inherits from the class `RealTimeThread`. In this way the scheduler is a real-time thread.

C. Dynamic Loading of Classes

Sometimes the server needs some class in order to satisfy the request of client. In this case the client must send all the classes needed by the server (e.g. `Scheduler` or

DeadlineHandler) Those classes are sent together with the client's parameters.

To implement this capability we have developed two classes to support the reception, and the loading of the classes needed by server. Those classes are: DynamicLoadClient to send the bytecodes of the needed class. The DynamicLoadServer class in server side is responsible for acquisition and storage of classes. Once the class is in the server it is loaded at runtime. To implement those capabilities we use two Java mechanisms: *reflectivity* and *dynamic load*. Thus, server can get information about classes and loaded them at runtime.

IV. Framework Qualities for Implementing Applications

We have defined the architecture of our framework with some qualities that can help us to implement distributed applications in a easy way and with different characteristics. Most of distributed systems are based on the client-server architecture. We have used this architecture to identify the components of our proposition. In figure 1, we present the architecture that we have defined to identify the relation between those both components of our framework. The architectures for client and server applications are presented next.

Client Applications

It is composed of three classes: Parameters, Client, andRemoteInterface each one of them has a specific role:

The Parameters class allows to encapsulate the scheduler name and time parameters to be sent to the server in order to control the execution of the application. This class has a data structure that can be modified or extended according to the the aplication needs. The *serializable* (java.oi package) interface has been implemented here to allow sending objects through the network.

The Client class is responsible of creating the connection with server. The main method uses the IP address and the number port for configure the socket, and it is also responsible for creating theParameters object. Once the communication is established theRemoteInterface class acts like a broker to make calls to the remote objects.

The RemoteInterface class inherits from theThread class, this means that the start() method creates the execution threads and calls the run() method in the Application class. To send the additional parameters, these can be defined as private attributes and can be initialized in the constructor. The RemoteInterface's constructor creates instances for the input-output flows for sending objects of type Parameters. Given that this class inherits from Thread class the run() method allows get several instances of RemoteInterface class executing in a concurent way.

Server Application

The server application is composed of the next classes: Server, RealTimeParameters and Application.

The constructor of Server class is responsible for establishing the mechanism for communication, i.e. creating the sockets. Server applications starts to execute through the main() method, waiting for client's requests, these are treated through the accept() method.

Once the server is running and accepting requests, it is necessary to create an instance of the RealTimeParameters class to get the parameters received through the input socket. Given that this class inherits from the Thread class a thread of execution is created for every client. In this way it is possible for the server to interact with several clients in a concurrent way. In server side the importance of this class is highlighted because it works as interface between Java standard and Java Real-Time. This interaction is obtained by using the javax.realtime package. The parameters received by server must be mapped into the real-time data types of the RTJS. This mapping is performed by using the classes Scheduler, PriorityParameters, RelativeTime, andPeriodicTime of RTJS.

After the communication is established and the parameters have been transformed into RTJS real-time formats, it is necessary to create an instance of the Application class, this class inherits from the RealTimeThread, so this is treated as a real-time thread with client's parameters associated.

Framework Qualities for Application Development

Our framework provides some facilities for application development, the aim for this is to help developpers in implementing distributed applications. These qualities are presented next.

Adaptability

It is possible to add additional mechanisms to our framework, in an easy and simple way. For example, replacing the standard scheduler with some specific implementation of a different scheduling policy. In our framework for modifying a component, like the scheduler, it is necessary to follows the next steps:

First, it is necessary to create the class for the new scheduler, the RTJS tells that the name of class must give an idea about the kind of policy being implemented. This class must inherit from the RealTimeThread class in order to implement the new scheduler as a real-time thread.

Second, it is necessary to use the LinkedList class (from the java.util package) to create a list of threads to be controlled by the scheduler. The constructor's class of new scheduler is responsible of: a) getting the first real-time thread and b) creating the real-time thread list. Once the list is created, threads call the startScheduler() method of the new scheduler in order to be executed according to the real-time parameters. Every application wanting to implement a specific scheduler must to follows these steps.

Dinamicity

However, it could be possible that the application needs some specific components, two situations could be possible:

1. The class file needed is in the server and client side.
2. The class file must be sent from the client to the server,

together with the parameters of the client.

Taking the example of scheduler, the client must send the name scheduling policy to be used by the server application together with his parameters. The server must compare the name of the scheduling policy passed with the name of the policy he knows. This is done using the `getPolicy()` (from the `Class` class) method to get the name of the new policy. If the name corresponds to the name it knows, then it can continue giving service to client. If names are different the file with the class must be requested by the server and then call the `startScheduler()` method to start the scheduler.

To be capable to localize the files classes, the server can use two mechanisms from Java Standard: the *reflectivity* and the *dynamic loading,* mentioned above. The server notifies to the client that it does not know the method needed by the client. The `RemoteInterface` class in client side must send the *.class* file to the server. This is achieved through the implementation of two classes the `DynamiLoadClient` and `DynamicLoadServer`. In the client side the `DynamicLoadClient` class is responsible for sending the bytecodes of the class file that the server does not know; it is sent through the socket as a vector of bytes. In the server side the `DinamycLoadServer` class is responsible for getting the file class sent by the client. Once the file is in the server side, it is loaded at runtime. In this way, it is possible to send files of classes between the sites cooperating in the distributed application. This facility gives to our framework the dynamic aspect for application implementation.

Versatility

With our framework, it is possible to implement applications within some of following modalities:

1. Application is implemented in the run() method of the `Application` class.

2. The `run()` method works as an interface between the `Application` class and the application where the service is implemented.

3. Creating an additional class in both client and server, where the data and methods are implemented.

An example of applications implemented in the first modality is shown in Fig. 4. In this case, the application is implemented in the `run()` method, this is the simplest way to develop an application. Through the `RealTimeParameters` class the server gets the parameters to be passed to the `Application` class, where the `run()` method implements the application. In the shown example, the application is a generator of Fibonacci numbers. The generation of numbers is made in a periodic way, and these are sent to the client according to the specified parameters.

Sometimes the application to be developed needs an additional class where the code for the service is specified. In this case the `run()` method of the `Application` class works as an interface to create instances of services, Fig. 5 shows an example of applications developed under this modality. The developed application is a generator of primes numbers. For

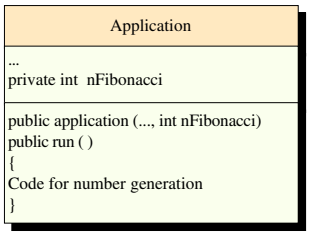

Fig. 4. The Application Class for Fibonacci

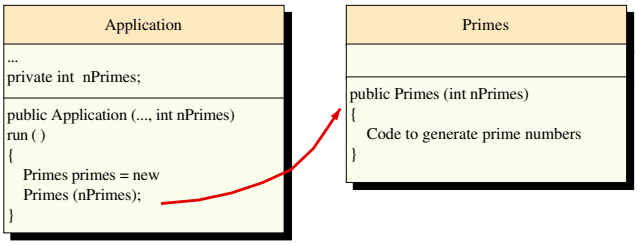

Fig. 5. Application and Primes Classes for Prime Number Generation

this application the`Primes` class implements the algorithm to generate the numbers. The `run()` method is used as an interface between the `Application` and the `Primes` classes where the constructor gets the client's parameters. The generated numbers are sent to the client according with its parameters.

In the third modality of development, an additional class is defined, where services are implemented. The application example developed within this modality carries out the comparison between two images, taken as integer numbers matrices. In this example the `Matrix` class implements the service. For this example we have implemented only one class, however several classes could be implemented if needed. As explained above if classes are not known by server, these could be requested to the client using the dynamic loading of bytecodes.

In this application, in the server side, the `Matrix` class carries out a preprocessing, which consists in reading the matrices and verifying their sizes. Once the preprocessing has been finished, they are sent to server together with the real-time parameters and the scheduling policy to be used. The `RemoteInterface` class sends all these data. In server side the matrices are rebuilt using the `Matrix` class, then the `RealTimeParemeters` class gets the matrices and other parameters (see Fig. 6).

Once the matrices are in the server site they are passed to the `Application` class where the `run()` method calls the `CompareMatrix` method in `Matrix` class, where the comparison of both matrices is achieved. The comparison is realized row by row in a periodic way.The resulting report gives the location and the values where the matrices are different.

V. Conclusions

Today, as the applications are more complex, it is needed to use several components to implement them, so distributed

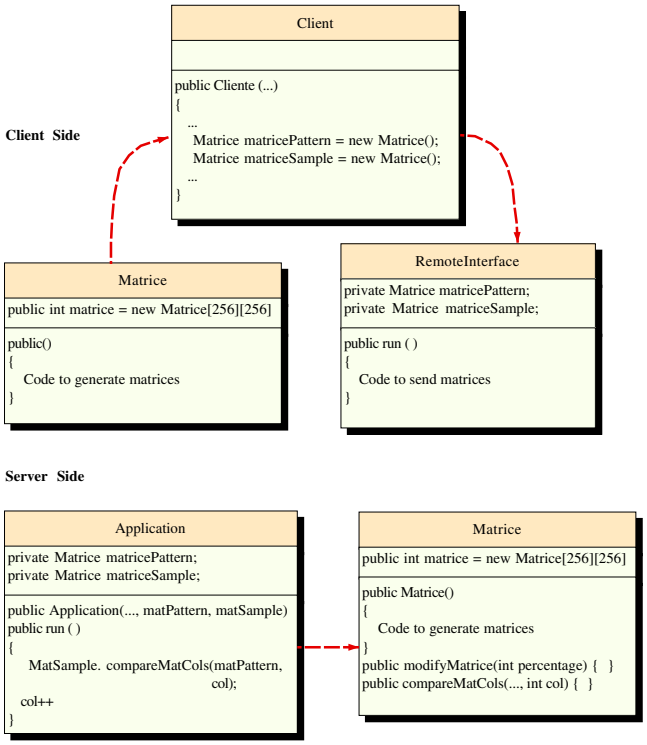

Fig. 6. Classes for Image Comparison

systems are the solution. In the sphere of real-time systems, the use of distributed systems is more requested every day.

Java is a promising technology for the development of real-time distributed systems thanks to some capabilities like simplicity, concurrency, and portability. However some problems are not yet resolved, like the definition of a RTSJ for real-time distributed systems. Very serious efforts have been carried out to define a real-time middleware using the Java RMI as the basic mechanism for communication.

In this work we present a proposition to propagate the real-time parameters and requests of scheduling policies between the components of a distributed system. With our proposition it is possible to respect the temporal restrictions in a distributed system.

We have proposed a Java-based architecture taking the Java-RMI as a basic development tool. Our solution is integrated with a JVM-RT, the API-Sockets and the reflectivity and dynamic loading capabilities of Java. With all these mechanisms, we have created a tool for the development of soft real-time distributed systems.

We have developed a generic framework with some additional qualities: flexible, adaptable, versatile, and dynamic.

Our framework is generic because it is possible to implement several types of soft real-time applications. It is flexible because the structure of classes can be adapted to the application requirements. It is versatile because we have implemented several scenarios of application. It is adaptable as it is possible to change some components of the structure (scheduler for example). Finally it is dynamic because it is possible to exchange the components needed for the service in runtime.

Our proposition also allows to reuse architectural components and implementation classes to facilitate the implementation of soft real-time distributed applications without starting from the scratch.

REFERENCES

[1] Bollella, G. and Gosling, J., "The Real-Time Specification for Java", *IEEE Computer Society,* vol. 33, num. 6, pp. 47-54, June 2000.

[2] Borg A. and Wellings A., "A Real-Time RMI Framework for the RTSJ", *In Proc. the 15th Euromicro Conference on Real-Time Systems - ECRTS'03,* pp. 238 - 246, IEEE Computer Society, July 2-4 2003.

[3] Dibble, P. C., *Real-Time Java Platform Programming,* Prentice Hall, First Edition, March 2002.

[4] Gomaa Hassan, *Designing Concurrent, Distributed, and Real-Time Applications with UML,* Addison-Wesley, July 2000.

[5] Grosso, W., *Java RMI,* O'Reilly, First Edition, October 2001.

[6] Henning, M. and Vinoski, S., *Advanced CORBA(R) Programming with C++,* Addison-Wesley Professional Computing Series, February 1999.

[7] Jensen, D. and Anderson, J., "Distributed Real-Time Specification for Java: A Status Report (digest)", *In Proc. of the 4th international workshop on Java technologies for real-time and embedded systems,* vol. 177, , pp. 3-9, Paris - France, October 2006.

[8] Kopetz, H., *Real-Time Systems Design Principles for Distributed Embedded Applications,* Kluwer Academic Publishers, Eight Edition, 2004.

[9] Oaks, S. and Wong, H., *Java Threads,* O'Reilly, Third Edition, September 2004.

[10] OMG (Object Management Group), *Real-Time CORBA Specification, Version 1.1,* formal/02-08-02, August 2002.

[11] Rusty, E., *Java Network Programming,* O'Reilly, Second Edition, August 2000.

[12] TimeSys, *Real-Time Specification for Java Reference Implementation (RTSJ RI),* http://www.timesys.com/java.

[13] Wellings, A., Clark, R., Jensen, D. and Wells, D., "A Framework for Integrating the Real-Time Specification for Java and Java's Remote Method Invocation", *In Proc. of the Fifth IEEE International Symposium "Object-Oriented Real-Time Distributed Computing",* pp. 13-22, Washington, DC., USA, April 29 - May 1st, 2002.

[14] Wellings, A., *Concurrent and Real-Time Programming in Java,* John Wiley & Sons, First Edition, September 2004.