ISSN 0280-5316
ISRN LUTFD2/TFRT--5756--SE

Portable Robot Control

Ferran Carlas Ponce

Department of Automatic Control
Lund Institute of Technology
June 2005

Department of Automatic Control **Lund Institute of Technology** **Box 118** **SE-221 00 Lund Sweden**	*Document name* MASTER THESIS *Date of issue* June 2005 *Document Number* ISRNLUTFD2/TFRT--5756--Se
Author(s) Ferrran Carlas Ponce	*Supervisor* Anders Robertsson at Automatic Control in Lund Klas Nilsson at Computer Science in Lund *Sponsoring organization*

Title and subtitle
Portable Robot Control (Implementering av realtidsplattform för robotreglering i Java)

Abstract

The topic of this master thesis is Portable Robot Control and it has been performed jointly with the thesis Portable Robot Programming. The robots mentioned in the title of the Master thesis are industrial robots. Industrial robots are embedded real-time systems. In the embedded real-time systems the computer is part of the system. As all other real-time systems they have to fulfill real-time requirements. Hence, they must be deterministic and predictable.Then, the control and programming tasks of these industrial robots must be performed with tools providing the mechanisms to fulfill the time requirements above-seen. The first task of the thesis is the communication of a robot system using a real-time network protocol. The real-time protocol chosen is ThrottleSim, the Java-based simulation of ThrottleNet. To perform this communications a real-time communications environment is designed. This environment is focused on the intermediate layer that is the link between the chosen network protocol and the application layer.
Once the communications in robot systems are implemented, they must be integrated in that system. A robot system consists of many different elements. These parts are the mechanical manipulator (robot) and the robot control system which can consist of the robot server, the computers where the control parameters are generated, the simulator or the network which is used for the communication between all the different parts.The implementation of a Java-based infrastructure to integrate all the parts of the robot system in some experiments is the second main-task of the thesis.
Finally, the tasks described above are implemented in the Java programming language, because of its wide range of advantages (platform independence, simplicity, security and robustness). Despite of the fact that Java has a lack of efficiency that makes it not suitable for real-time systems, it is possible to use it as a real-time language through the Java-to-C compilation. This solution provides the advantages of Java as a programming language and solves its main disadvantage.

Keywords

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title 0280-5316	*ISBN*

Language English	*Number of pages* 74	*Recipient's notes*
Security classification		

Contents

List of Figures

List of Tables

1 Introduction

The topic of this master thesis is Portable Robot Control and it has been performed jointly with the thesis Portable Robot Programming [1].
The work has been carried out within the Erasmus exchange program at the Departments of Computer Science and Automatic Control, Lund Institute of Technology, Lund University, Sweden.

Industrial robots are embedded real-time systems and, like all other real-time systems, they have to meet time requirements. These time demands can be harder or softer depending on the task. Thus, the tools chosen to program and control these robots have to be chosen carefully.

Platform independence, simplicity, security and robustness are some of the Java programming language advantages. The weak point of Java is related with the fact that Java is not explicitly intended for real-time applications. Therefore, it has a lack of efficiency due to the virtual machine and the automatic garbage collection. However, research at the Department of Computer Science has been working in order to solve these problems and to use Java as a real-time language. The solution chosen has been compiling Java to the C language.
Seizing the advantages that the Java language provides and improving its weak spots, a reliable base to set up a real-time communications environment it is achieved. Furthermore, a portable infrastructure to establish real-time communications is performed as a first task of this master thesis.

A robot system consists of many different elements. These parts are the mechanical manipulator (robot) and the robot control system which can consist of the robot server, the computers where the control parameters are generated, the simulator or the network which is used for the communication between all the different parts. All the elements listed above might work on different machines, platforms, programs and so on.
The second task of this thesis consists of integrating all the elements of a robot system in some experiments.

All the desired experiments could not be carried out during the period of time this master thesis took. Thus, there is place for future work within this area.

1.1 Outline of the Report

Chapter 2 – Background
In this chapter three general concepts strongly related with the development of this thesis are defined. These concepts are Real-Time Systems, Java in Embedded Real-Time Systems and Portability. Also, some related work used in the project is explained. Finally, the concept of serialization is described accurately.

Chapter 3 - Real-Time Communications
This chapter deals with the design and implementation of all the Java packages built to perform a real-time communication.

Chapter 4 - Real-Time Communications Experiment
In this chapter the test experiment of the real-time communication packages, implemented in Chapter 3, is presented. Also the differences between real-time and non real-time transmission are studied.

Chapter 5 - Robot Experiments
This chapter introduces the experimental platforms. Different experiments performed with the robots or the Java3D simulator are described. These experiments integrate the real-time communications part, the trajectory generation for industrial robots and the real-time execution and simulation.

Chapter 6 – Conclusions
In this chapter the thesis is briefly summarized and the final conclusions are presented and discussed. Finally, the proposed future work is listed.

2 Background

This thesis deals with many fields and concepts related with Automatic Control and Computer Science. The concepts described in this chapter can be divided into two groups. The first one includes general concepts strongly related with the thesis, i.e., the concept of real-time systems or the concept of serialization. The other group are the tools used to perform the work, i.e., network protocols like ThrottleNet and the Java to C compiler.

2.1 Real-Time Systems

A real-time system is *any information processing system which has to respond to externally generated input stimuli within a finite and specified period* [Burns and Wellings, 1990].
The most important characteristic of a real-time system is that the system has timing requirements that must be met. This leads to the statement that the systems must be deterministic and predictable. It is important to know the time it will spend at every task and its priority in order to schedule them and fulfil the time requirements.

The system to consider in this thesis will be an industrial robot manipulator with the corresponding open robot control system. Industrial robots are real-time embedded systems. They can be both *hard real-time and soft real-time systems*. A hard real-time system can be defined as a system where it is absolutely imperative that the responses occur within the required deadline. While, in soft real- time the system still functions if the deadlines are occasionally missed.

2.2 Java in Embedded Real-Time Systems [2]

Java is an object-orientated concurrent programming language developed by Sun Microsystems in 1995. Java is simple, robust and secure language. In order to increase its portability, Java is an interpreted language. The byte code generated by the compiler is interpreted by the JVM (Java Virtual Machine). Thus, Java is platform independent; it can be executed on any computer, as long as it has a Java Runtime Environment (JRE).
Java is not the ideal real-time language; it was not explicitly intended for real-time applications. The main problem is the lack of efficiency due to the stack-based virtual machine and the automatic garbage collection. The automatic garbage collector makes Java execution non deterministic.
Compiling Java to native code is the solution chosen (Department of Computer Science, Lund Institute of Technology) in order to make Java a viable language programming for hard real-time systems.
In order to perform this compilation, a Java to C compiler built in the Department of Computer Science has been used [3].

2.3 Java to C Compiler [3]

The Java to C compiler translates Java class files into C source code. This allows the construction of directly executable C-programs.
The compiler was built using the *JastAdd* compiler construction toolkit [4].
In Figure 2.1 the process that Java files follow until the execution is explained. The first step is the translation of the Java files to the corresponding C files (.c and .h). The second step is the compilation and linking of the C files and the native methods jointly. This stage is carried out by the C compiler. Once everything is compiled successfully, the file is runnable.

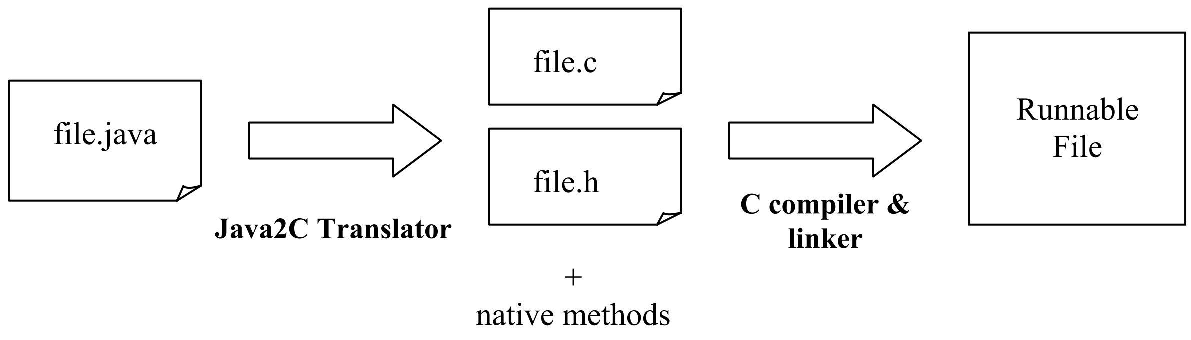


Figure 2.1: Process to convert Java files to runnable C files

2.4 Portability

Portability refers to the ability to run a program on different machines. Running a given program on different machines can require different amounts of work (for example, no work whatsoever, recompiling, or making small changes to the source code) [5].
Java is platform independent; therefore it is possible to run Java programs in any machine (independently of the CPU and OS).
Thus, in order to have a portable system, Java has been used as a programming language as well as other Java-based tools like Java3D Simulator (Chapter 5) or a Java to C compiler.

2.5 ThrottleNet [6]

ThrottleNet is a real-time network based on fast-switched Ethernet, where every node is connected to each port of the switch using full duplex. ThrottleNet provides a good management of the real-time data transmissions and solves some problems that standard protocols (TCP or UDP) have.

ThrottleNet is a real-time layer over Ethernet to fulfil real-time requirements.
ThrottleNet can run in two different modes: centralized and decentralized. In centralized mode there is a central node, called GlobeThrottle, which controls the traffic in the network. All the connections pass through GlobeThrottle. In the decentralized mode there is no central node, thus, all the connections are controlled by the nodes involved in these connections. In this thesis the centralized version will be used.

All the communication packages designed in this thesis use ThrottleNet or its Java-simulation, ThrottleSim, as a network protocol.

2.6 ThrottleSim [7]

ThrottleSim is the Java-based simulation of ThrottleNet. To simulate the ThrottleNet protocol, ThrottleSim uses User Datagram Protocol (UDP) due to the similarities to ThrottleNet.
The main goal of the procedure is to avoid the network congestion so that the time restrictions are fulfilled.
To implement our real-time communication packages we have been working with ThrottleSim instead of ThrottleNet. Working with the simulated protocol is easier because the appropriate device drivers and administrator access to the system are not required. Thus, it is possible to work always in user space.

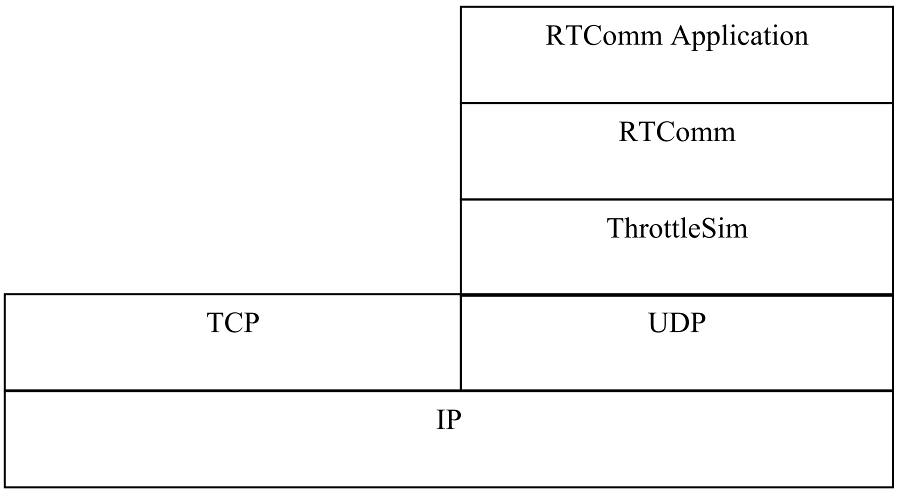


Figure 2.2: ThrottleSim over UDP

Figure 2.2 shows that ThrottleSim works upon UDP and the RTComm package (3.2) does the same on ThrottleSim. Finallay, on top of RTComm, all the applications are implemented.

2.7 Serialization [8] [9]

The concept of serialization is very important in order to understand the reason of the implementation of some packages that have been designed and, at the same time, is the link between them.

Before talking about serialization withyin the project, the theory of serialization and externalization is explained

Serialization is the process of reading or writing an object from or to a stream of bytes. The goal of this process is to have the representation of this object as group of bytes. Once the object is represented as a series of bytes it opens many possibilities of handling these data. For instance, in this project serialization is used to store and transmit the serialized data.

The process of serialization has two parts, we will call them *serialization* and *deserialization*.
Serialization consists of writing the object to an ObjectOutputStream using the method writeObject of the OutputStream class.
Deserialization is the opposite process to serialization. Deserialization reads the serialized object from an ObjectInputStream using the method readObject of the InputStream class.

ObjectOutputStream and ObjectInputStream must be constructed based upon another stream like a pipe, a file or a byte array. In this project PipedStreams are used to build the ObjectStreams because they provide good performance moving data between threads.
Therefore, ObjectOutputStream is constructed on a PipedOutputStream and ObjectInputStream is constructed upon a PipedInputStream.

Objects can be serialized if they implement the java.io.Serializable interface. It is an empty interface, because it does not declare any method or field. It just identifies objects that can be serialized or deserialized.

Externalizable is a subinterface of Serializable.
Sometimes it is possible or desired to control the serialization process. In these cases the interface Externalizable is used to give the control of the process to the object itself. To perform this control over its serialization the object must implement the methods writeExternal and readExternal.

```
package java.io;
public interface Externalizable extends Serializable {
    public void writeExternal(ObjectOutput out) throws IOException;
    public void readExternal(ObjectInput in) throws IOException,
                                java.lang.ClassNotFoundException;
}
```

In the part of code written above the headers of the methods that an Externalizable object must implement are shown.

Using the Externalizable interface the serialization process is manual. It means that the content of the object has to be known in order to implement the writeExternal and readExternal methods. These methods will be implemented depending on the type of data that is serialized (primitive types, objects, strings or arrays).

The difference between the Externalizable and the Serializable interfaces is the information contained in the output (serialized stream). In the externalization process some type information is avoided, it leads to a more compact output.
Therefore the Externalizable stream structure will be shorter, but it will contain less information. Thus, if the final goal of the serialization is the transmission of information, an externalized stream will be transmitted faster than a serialized one.

2.7.1 Serialization in the project

Object serialization is very important to understand and link the different parts of this project. Serialization is the link between the information to be sent and the network protocol used to send this information.

If the information to be sent is an object, a matrix for instance, and the network protocol transmits bytes, there will be something missing in between. Serialization is the mechanism that will fill the gap in between converting objects to byte streams and vice versa.

LabComm is a package that provides classes and threads to implement and support the serialization process.
The other package related with serialization is the DataStructure. The different classes belonging to this structure implement Serializable or Externalizable. The ones implementing Externalizable provide the methods writeExternal and readExternal on its own code. These two packages will be described in detail in Chapter 3.

3 Real-Time Communications

To transmit data in real-time not only a real-time network protocol is required, but a real-time communication environment is needed.

3.1 Real-Time Communication Environment in Java

A real-time communication environment consists of many different elements:

Starting from the lower level, the first element is the **network protocol**. A real-time network must fulfil the time demands on network data transmission. As mentioned before, ThrottleNet (ThrottleSim) is the protocol chosen in this work.

On the upper level, the **application** is another element belonging to the communication environment. Dealing with real-time transmissions the application will establish connections and send, receive and handle data. Different applications have been implemented during the project. A general description of their structure and working procedure is carried out in Section 3.5.

In any transmission, one of the most important parts is the transmitted data. Working in a real-time environment the information to be transmitted might be very specific. Therefore, this information can be organized and encapsulated in a **data structure**. Just as most of the data sent in real-time is very specific, general data should be possible to be transmitted as well. Thus, the data structure may be flexible. This master thesis contains the suggested implementation of a data structure (DataStructure package, Section 3.3).

To link the elements described above (network protocol, transmitted data and the application) an **intermediate layer** has been created. Thus, the user (application) does not deal with the network protocol directly. This intermediate layer called RTComm provides the strictly necessary methods to perform either a real-time or non real-time communication.

Within this environment other elements supporting data conversion and communication between layers are necessary. All these methods are included in the package called LabComm.

All the elements are implemented in the Java programming language.

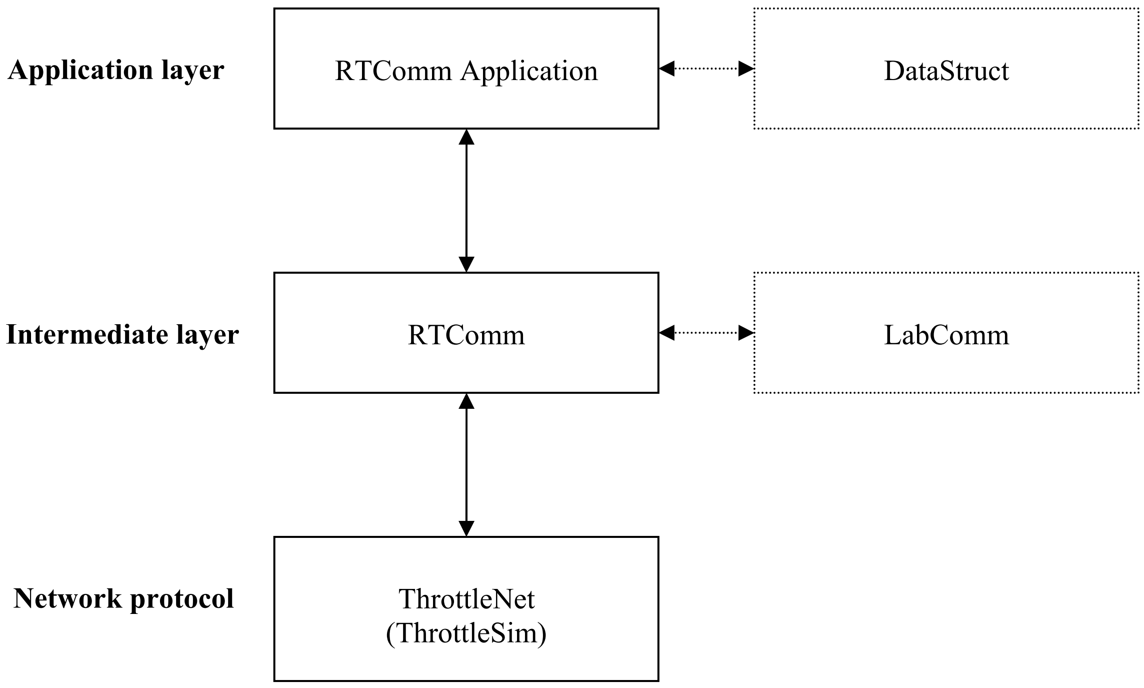


Figure 3.1: Elements of a real-time communication environment

In this thesis all the environment described above, except the network protocol, has been implemented. The results are three packages (RTComm, DataStructure and LabComm) and several applications. Everything is based on the ThrottleNet (ThrottleSim) protocol. The implementation of theses three packages is described below.

3.2 RTComm package

The RTComm package lays over the ThrottleNet networking protocol.
RTComm comes from real-time communication, but this package is not only made to deal with real-time communications, it can also deal with non real-time communication. Therefore, RT means, in this case, that this package can support both real-time and non real-time communications.

The socket class is the root of the package. It is an abstract class that provides the common implementations to establish a connection (socket) at this level, either in real or non real-time.

Below the two different branches of the socket class can be distinguished; the real-time and the non real-time branch, see Figure 3.2.

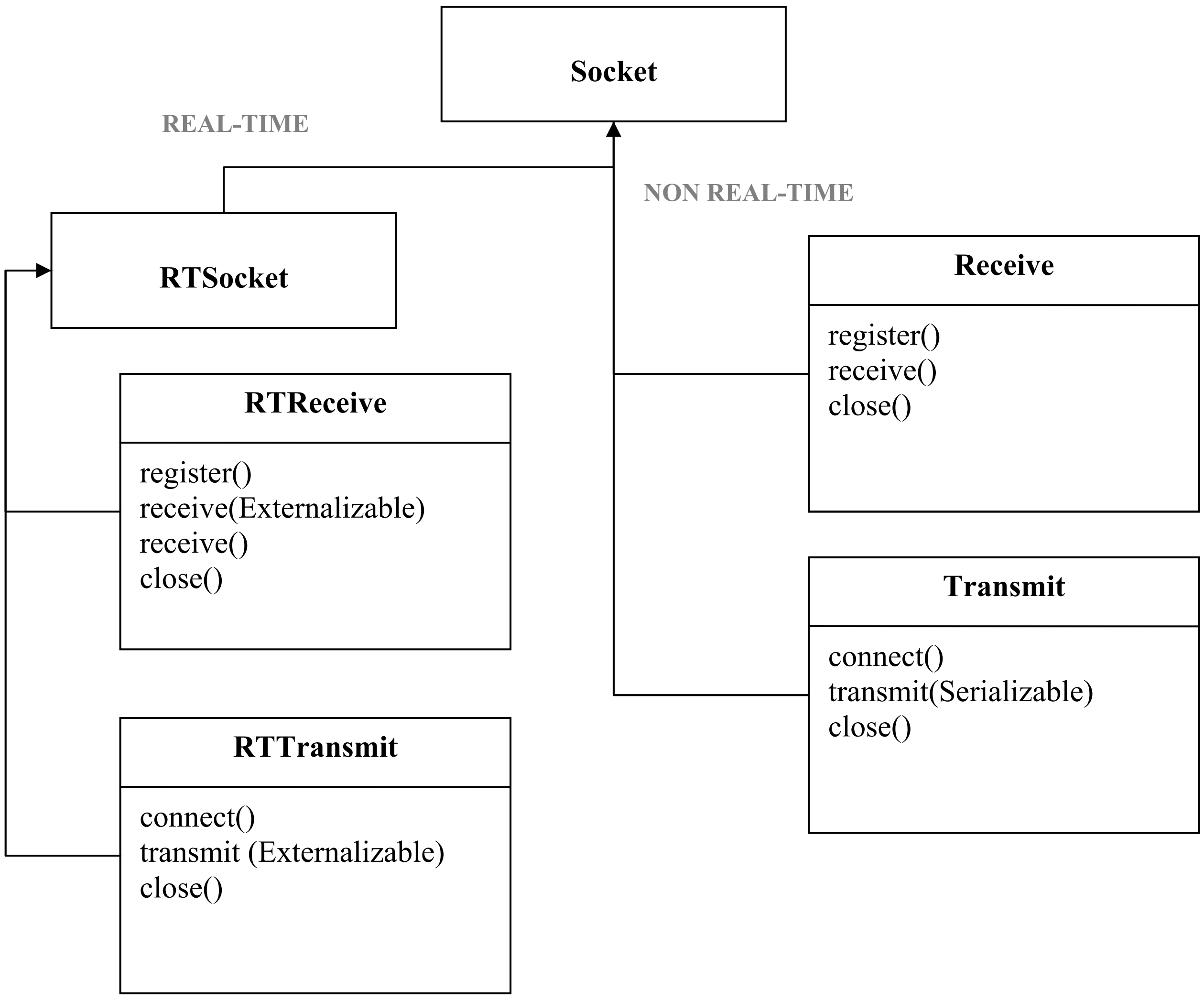


Figure 3.2: Structure of the RTComm package

3.2.1 Real-Time

On the real-time side, another superclass is implemented. RTSocket is the superclass for the real-time classes. It provides the implementations, not included in socket class, to establish a real-time socket.
Once the superclasses are described, the real-time part of the package contains two classes.
The RTReceive class describes an intermediate layer between the application and the ThrottleNetReceiveSocket [7]. The class implements the following methods:

- *register()*. Registers the receive socket.
- *receive(Externalizable)*. Receives an Externalizable object as a byte array and returns it as an object in its original shape.
- *receive()*. Receives a Serializable object as a byte array and returns it as an object in its original shape.
- *close()*. Closes the connection on the receiver side.

The other class of the real-time side is RTTransmit. It describes an intermediate layer between the application and the ThrottleNetTransmitSocket [7]. The class implements the following methods:

- *connect()*. Opens a specified connection in ThrottleNet.
- *transmit(Externalizable)*. Externalizes and transmits through ThrottleNet an Externalizable object.
- *close()*. Closes the connection on the transmitter side.

Only one receive and transmit method will be used depending on the type of object and the choice of the user. An Externalizable object can be serialized or externalized, while a Serializable object always will be treated as itself.
Comparing with Throttlenet, Socket and RTSocket would overload ThrottleNetSocket, while RTReceive and RTTransmit would do the same with ThrottleNetReceiveSocket and ThrottleNetTransmitSocket.
RTReceive implements almost the same methods as ThrottleNetReceiveSocket, but ThrottleNetReceiveSocket works on a lower level than RTReceive. It means that inside the methods of RTComm the methods of Throttlenet package are called. The same happens with RTTransmit and ThrottleNetTransmitSocket.

3.2.2 Non Real-Time

For the non real-time side the only superclass is the Socket class. The non real-time branch has two classes: Receive and Transmit.
The Receive class has the same methods as RTReceive, but it can only receive Serializable objects
The Transmit class also has the same methods as RTTransmit, but it can only transmit Serializable objects.
The others differences between the classes in real and non real-time are the parameters required to establish the connections. This topic will be widely explained at the end of this chapter.

For the non real-time part, Socket would overload ThrottleNetSocket [7], receive would do the same with ThrottleNetReceiveSocket and transmit with ThrottleNetTransmitSocket.

Why not Externalizable methods?

Because there is no guarantee that the data structure to be sent in real-time implements Externalizable, but this data structure will always be Serializable by nature.
It could be that an Externalizable object is sent using a non real-time connection. In this case, the object would be transmitted and received as a Serializable as well.
Externalizable is a subclass of Serializable, therefore there is the possibility of using the same methods. The advantage of the externalization is its compactness, but in non real-time, without temporal restriction, it becomes a characteristic rather than an advantage.

Anyway, the main reason not to use Externalizable methods in non real-time comes from the data structure point of view.

For the data structure designed and used in this project there are classes to work in real-time (LabObject) and classes to work in non real-time (GeneralObject). The GeneralObject class is designed to work in non real-time with java.lang.objects. This class implements Serializable, since it is open to all kinds of objects without specification.

3.2.3 Real-time and non real-time parameters [7]

To establish a ThrottleNet connection some parameters are requested. Some of these parameters are different depending on the type of connection.
There are some parameters that are required in both types of connection, real-time and non real-time:

name: uniquely identifies the specific connection between a sender and a receiver.
size: defines the maximum amount of data (bytes) allowed in each packet transmitted. The packet size is an essential parameter, but the way to calculate this size in bytes is different depending on the type of connection.
period: defines the minimum time in microseconds (µs) between two packets transmitted on the network.
type: defines, if the connection will be real or non real-time.
dead_line: defines the maximum time to deliver each packet in microseconds (µs). It is a parameter uniquely defined by the receiver in the registration process.

3.2.4 The *size* parameter

In the ***real-time*** transmission the packet size is the size of the largest object that can be sent fulfilling the real-time restrictions. The largest object is built depending on the features of the system.
For example, the system we are working on is a robot with 5 motors. To control the motor, we need to give to the robot three values per motor. Thus, the largest object would be the size of the structure encapsulating these 15 values (5x3) that we send to the robot.
The size of this largest object is calculated serializing it into a stream of bytes. The object is serialized and not externalized because it wants to know the maximum size. The serialization process is less compact than externalization; therefore serialization is the worst case possible (maximum size).

Non real-time connections do not have to fulfil temporal restrictions. Therefore, the packet size will have another meaning. In this case it can be call *minimum effort*. It is the minimum object that can be sent. Thus, it assures that the minimum object can be sent and it protects the real time connection preventing the overloading of the net. The way to obtain the size is exactly the same as in the real-time transmission.

3.2.5 Non-real time traffic and ThrottleSim

ThrottleSim was not ready to handle non real-time traffic when we started our project. The main idea to handle this non real-time traffic is to dedicate part of the bandwidth of the connection to it.
Different solutions were possible. The implemented solution consists of
allocating the bandwidth in a static way. The main idea is to consider real and non real-time connections belonging to the same IP address as independent connections, with different network features.

Changes in the implementation of ThrottleSim were:

- Introducing a new connection parameter *type*. This parameter has to be handled exactly like the old parameters. It means that the parameter will be saved in the structure of GlobeThrottle where the network parameters are saved and can be checked. It implies a modification of GlobeThrottle.

- New calculation way of the available bandwidth for the connections. Before the calculation of available bandwidth was made according the formula:

Available Bandwidth= Max bandwidth-busy bandwidth;

Now, a correction factor is applied to devote one part of the connection to non real-time. The available bandwidth is calculated as follows:

Real-time *Available BW= (1-book percentage)* Max BW - busy BW*

Non real-time *Available BW= (book percentage)* Max BW - busy BW*

The *book percentage* has values between 0 and 1. Presently it is set to 0,1 (10%). But, it can be modified depending on the proportion of real and non real-time connections. At the moment, the process to allocate bandwidth is static; to change the percentage a parameter of GlobeThrottle class has to be changed. In further implementations a dynamical bandwidth allocation would be better.
The *Maximum bandwidth* is 100Mbits/s.
The busy bandwidth will be the sum of the bandwidth of all connections with the same IP address.

Most of the parameters are influenced by the value of the available bandwidth. Thus, this modification in the way how to calculate the bandwidth will have effect on the values of the other parameters.

3.2.6 Transmission Procedure

The process of transmission is strongly related with the lower level of the communication environment, ThrottleNet (ThrottleSim).

These are the steps followed in a transmission:

register() - the receiving node creates a ThrottleNetReceiveSocket and the information required to register the new connection is sent to GlobeThrottle. GlobeThrottle extracts this information and saves it. GlobeThrottle answers by sending a temporary value of the allowed fragment size needed to initiate the communication between two nodes.

connect() - The transmitting node creates a ThrottleNetTransmitSocket and the request to participating in a specified connection is sent to GlobeThrottle. GlobeThrottle saves the received information and checks the correctness of the parameters. Finally it returns a reply with the necessary data.

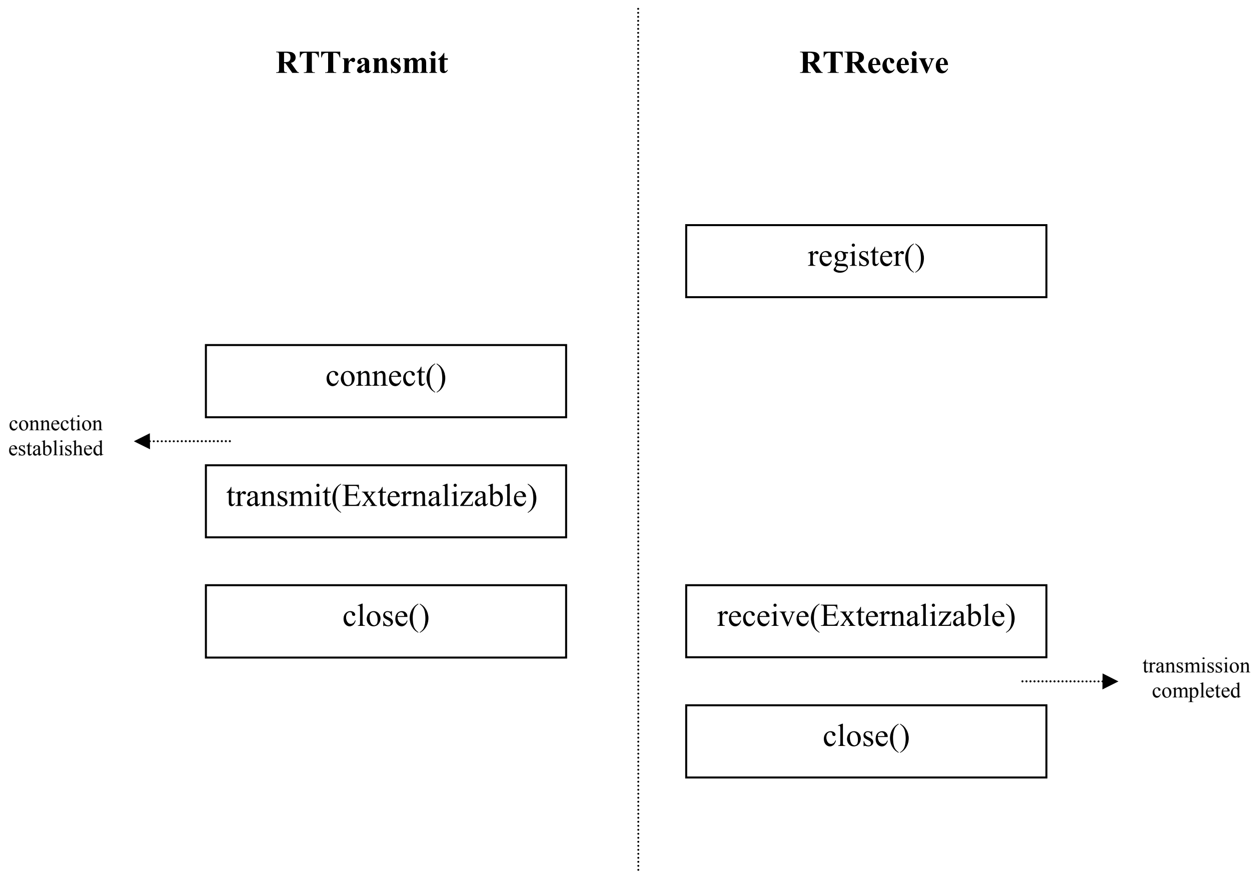


Figure 3.3: Calls to the required methods to perform a real-time transmission

transmit() - the process can be summarized in two steps: conversion of object to byte[] and sending of this byte[]. Depending on the type of connection the conversion process will use the LabComm.ExternalComm or LabComm.SerialComm packages (3.4). After the conversion, the data will be sent using ThrottleNetTransmitSocket.transmit(byte[]).

receive() - This process is the same as the transmit process but in the reverse direction. Firstly, data is received using ThrottleNetReceiveSocket.receive(byte[]) and, after that, the byte[] is converted into object again.

close() - The close procedure calls the close methods of ThrottleNetReceiveSocket and ThrottleNetTransmitSocket. It does not matter in which order the nodes close their socket. The nodes send the request array to GlobeThrottle, with the REMOVE command followed by the connection name. The central node removes the connection information from its internal network database.

3.3 DataStructure package

DataStructure is the Java based data structure used in this project. The implemented data structure has its origin in the packages MatComm [10] and LabComm [11].
The first version of the structure was called MatComm. It provided Matlab matrices of different types for communication. These matrices can be converted to streams in order to be sent after that.
The second version of a Java based data structure is called LabComm. LabComm extends the structure providing the superclass LabObject and other structures more complex than only matrices.
LabComm already used the concept of serialization to convert Java objects to streams due to the java.io.Serializable interface.
LabComm is the starting point for the new Java based data structure.

3.3.1 New Java-based data structure

The superclass of DataStructure is an abstract class called LabMessage. Message means something to be sent without specifying what kind of data.
LabMessage is the superclass for LabObject and GeneralObject.
LabObject is the structure used to send a specific kind of object (like different types of matrices or any structure whose content is known) in real-time. LabObject has two properties:

- *Non Abstract in the low level*. The lowest element of any branch of the structure has to implement Externalizable. It means that data stored in the structure is known and methods writeExternal and readExternal are implemented in the code.
- *Sized*. There is a maximum size to fulfil the temporal restrictions fixed by a LabObject. This second property comes from the first one. Since the structure is defined until the lowest level, the user can know the largest object of that type that it can be sent to fulfil the real-time restrictions (for instance, a matrix of chars with 20 rows and 20 columns).

LabObject is a structure ready to work in real-time, but it can work in a non real-time connection as well.
Below LabObject, more specific classes are defined.
LabMatrix is a package that includes matrices of chars, doubles, floats, integers and strings.
LabStructure provides classes and methods to build more complex structures combining the different types of LabObject.
Other classes could be created inside LabObject, for instance LabData.
LabData is a class that provides a structure to send log data like measurements from different sensors with corresponding time stamps etc.

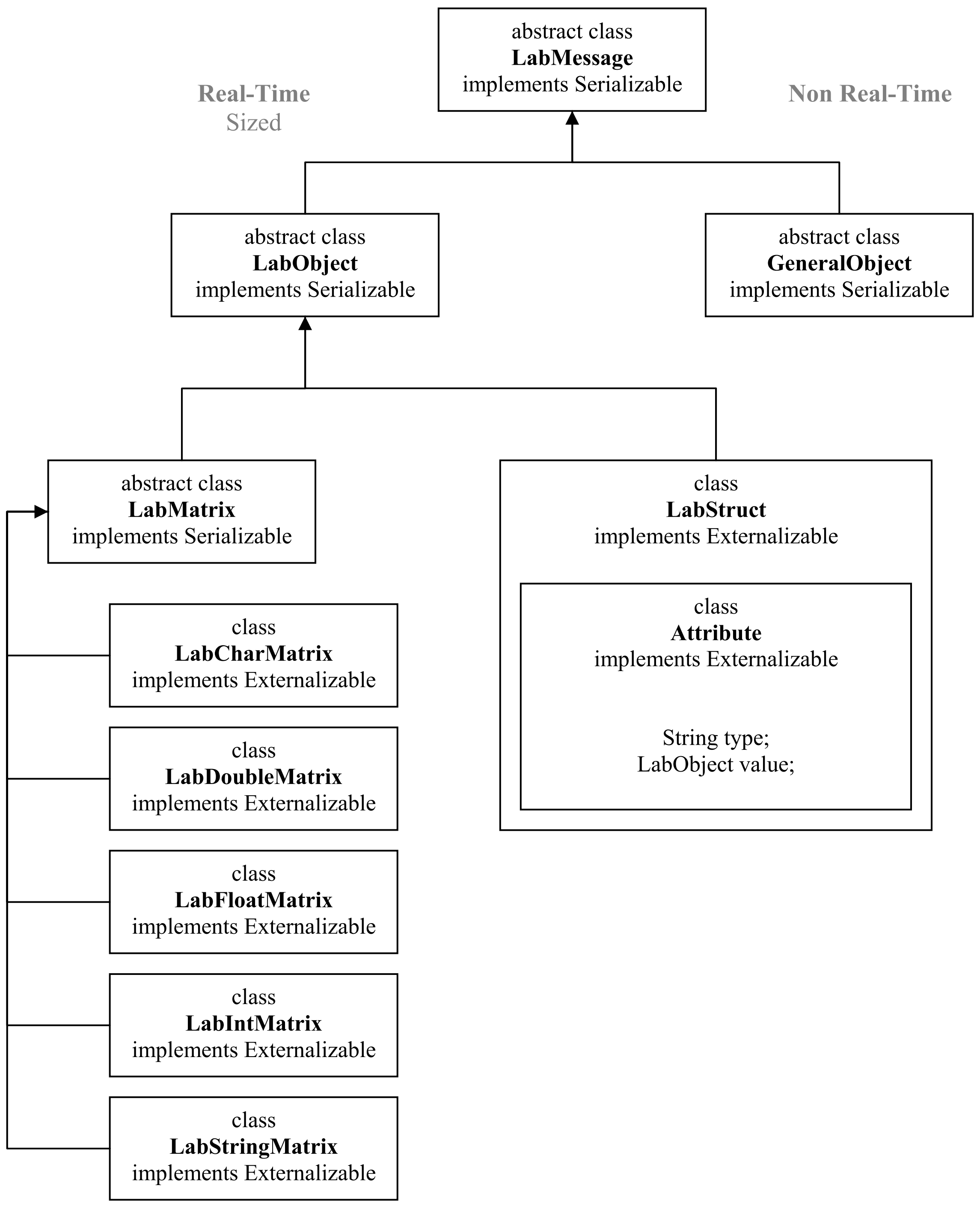


Figure 3.4: The implemented data structure

At the same level as LabObject there is GeneralObject. GeneralObject is the abstract class to define Java.lang.object without any other specification.
It is made to work with objects without temporal restrictions when they are transmitted.
GeneralObject is also the superclass for further more specific objects.

All the abstract classes implement Serializable. The abstract classes are empty and implementing Serializable it is not necessary to implement read and write methods, the default methods writeObject and readObject are used instead.

The non abstract classes implement Externalizable. Thus, the methods writeExternal and readExternal are provided by the structure.

3.4 LabComm Package

The package LabComm lays between the application level and the network protocol (ThrottleNet). It has two subpackages: SerialComm and ExternalComm.
SerialComm is the package that provides the classes, methods and threads to deal with the Serializable objects and the serialization processes, while ExternalComm does the same with the Externalizable interface.

3.4.1 Detailed description of LabComm:

- Declares, builds and initializes the buffers, PipedStreams and ObjectStreams required to perform the serialization and externalization processes.

- Provides the methods to serialize and externalize objects in both directions (object to byte array and byte array to object).

- Provides the threads to read from the corresponding pipe with the serialized or externalized bytes at the same time as these processes are being performed.

- Provides the threads to write to the corresponding pipe the byte array to be converted to an object (serialization or externalization) at the same time that this process is being performed.

3.4.2 Why readThread and writeThread? [9]

When an object is serialized or externalized in whatever direction (object to byte array or byte array to object), ObjectOutputSteram is used to write the bytes to, and ObjectOutputStream is used to read the bytes from.
Data is written in the ObjectOutputStream in block-data records. The block factor used for a block-data record will be 1024 bytes. This block factor produces a size limitation of the serialized objects. The largest object to be serialized at the same time is 1024 bytes.
The size limitation of the whole process may not come from the serialization process. It may be ThrottleNet which is the one introducing size limitation in the transmission instead of the internal processes.

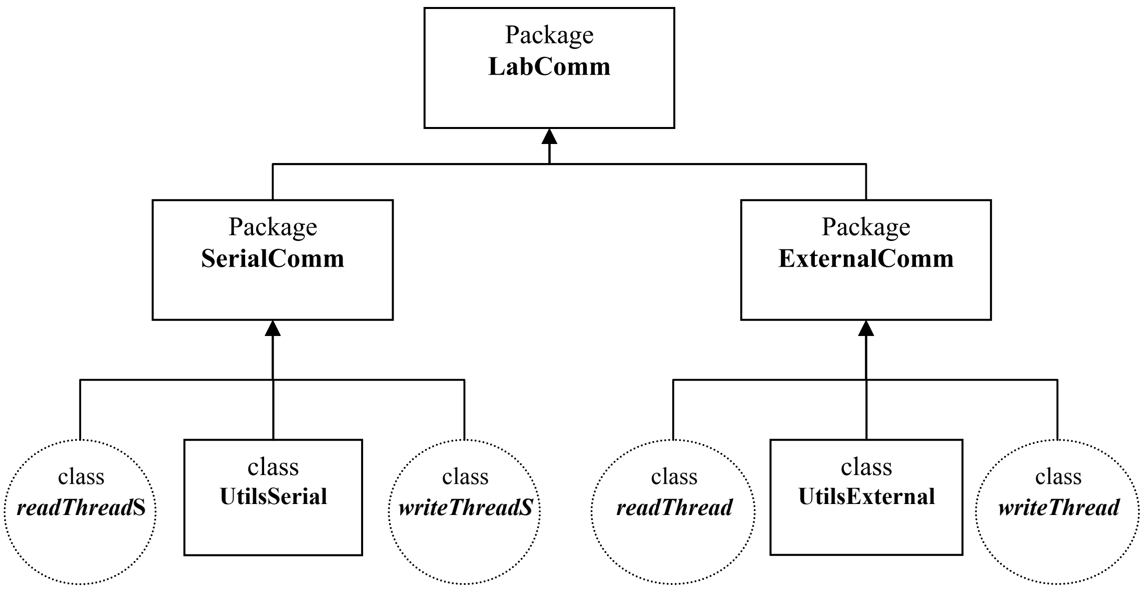


Figure 3.5: Structure of the LabComm package

Therefore, to avoid this size limitation two threads are inserted in an intermediate layer. *readThread* and *writeThread* are two intermediate threads performed before (*readThread*) and after (*writeThread*) the transmission and outside ThrottleNet.

At the same time that the object is converted to a stream of bytes, *readThread* reads these bytes and stores them in an auxiliary buffer. After performing the transmission, the received data is written to the ObjectOutputStream by the *writeThread* at the same time as the reconstruction of the objects is being done.

Without these threads the whole process could be performed, but large amount of data could not be treated.

The only disadvantage of the inclusion of these threads is that they may consume some extra time that sometimes could be saved without them. For instance, everything could be performed without these threads if the data to be sent is less than 1024 bytes. But, in this case, another thread may be used to know previously the amount of data to be sent.

However, the advantages of the inclusion of these threads are more remarkable:

- Many small objects can be sent in the same transmission.
- Large objects can be transmitted.

3.5 RTComm Applications

The last element missing to complete the description of the real-time communication environment is the one located on the upper layer, the application. Applications of this package will be called RTComm Applications.
The application described below is the first step to integrate the RTComm package in a platform either simulated (Java3D simulator) or real (robot).
In Figure 3.6 there is a description of the application's working procedure.
On the upper level the main application creates the object belonging to DataStructure. After, it starts the TransmitterThread and the ReceiverThread. It is also possible to call all the methods from the main application, but using separate threads is easier to divide the transmitter side from the receiver side.
After establishing the connection, the transmitter thread calls the RTComm methods to transmit the desired object. In the RTComm level, located between the application layer and the networking layer, the externalization process is carried out with the support of LabComm. When the information is encapsulated in a byte array, RTComm calls the ThrottleNet methods to transmit the data.
On the receiver side, the data is received using the ThrottleNet methods. After, RTComm calls the LabComm methods to perform the process of deserialization. When the process is finished the information is encapsulated again in a DataStructure object as the original one.

3.6 Summary

The real-time communications environment composed of three packages and based on a real-time network protocol has been implemented. The code is attached in Appendix 1. Creating an application to transmit real-time data is extremely easy and versatile using RTComm.

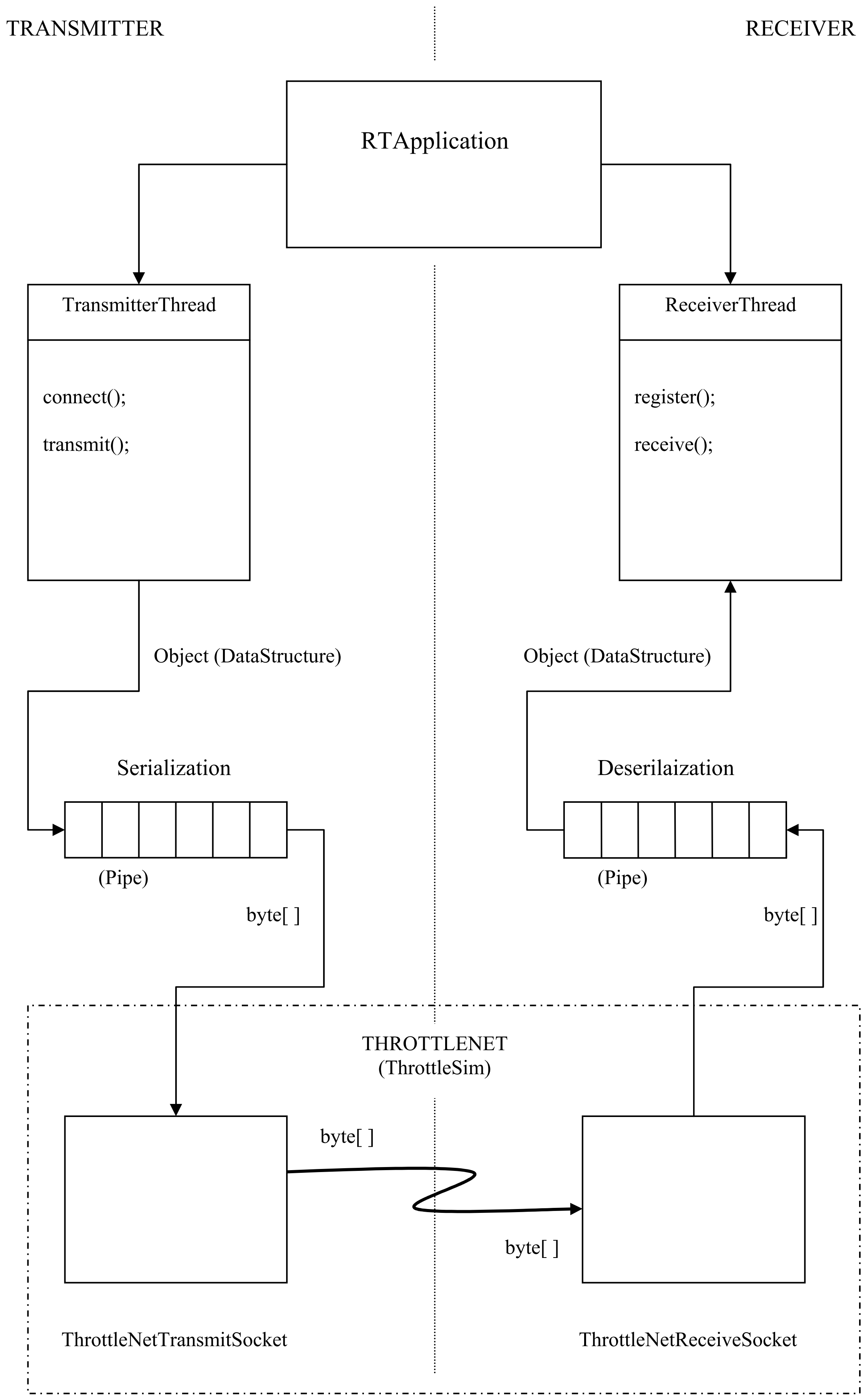


Figure 3.6: Working procedure of a standard RTComm application

4 Real-Time Communications Experiment

So far, several applications have been implemented and tested on the same computer. But, the aim of the communication environment is to communicate between different computers. Thus, this experiment performs this connection between two computers.
Before beginning to describe the experimental setup, its goals will be stated:

- Test the correct behaviour of the ThrottleSim protocol on a real platform.
- Evaluate the utility of the communications environment implemented in this thesis project.
- Bring to the user the possibility of setting the connection properties through a graphical interface.
- Analyze the time-differences between a real-time and a non real-time transmission.

In the experiment two computers are connected through Ethernet. One computer launches GlobeThrottle and the receiver thread, while the other computer launches the transmitter thread.
In order to make the experiment more realistic, the transmitted data are joint positions of the IRB6 or IRB2000 robots. The receiver thread is a Java3D Robots application that will run once the data are received, see also [12].

Experimental equipment:

	Node1	**Node 2**
Role	Receiver Thread (Java3D application) + GlobeThrottle	Transmitter Thread
CPU	AMD ATHLON XP 2000+ 1.67GHz	Intel Pentium II 350MHz
Memory (RAM)	512MB	128MB
Hard disk	80GB	20GB
Operating System	SuSE Linux	Debian Linux

Table 4.1: Characteristics of the experimental equipment

The network connection used is an Ethernet connection with 100Mbps of speed.

RTComm provides the possibility of setting some connection parameters. Some of these connection parameters can be set by the user through a GUI (Graphical user interface). The connection settings are: name of the connection, type of connection and period time. Also, the robot simulated in the Java3D application can be selected (IRB6 or IRB2000). The robot chosen is related with the connection name.
Once all the parameters are set, pushing the accept button the connection is established and the application is launched.

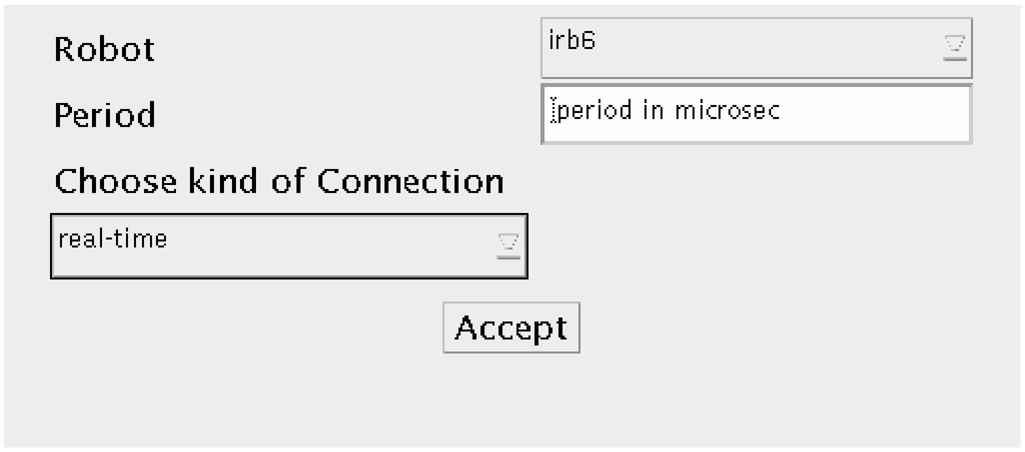


Figure 4.1: Graphical User Interface of the experiment

This tiny interface is the easiest way to show the versatility of the RTComm package, because it provides to the user simple and useful options and it prevents him or her to deal with the trickiest parts of the transmission procedure.

4.1 Time tests

In the experiment different amounts of data have been transmitted either in real-time or non real-time. The time tests have been performed without the GUI interface and the Java3D simulator applications, because the goal of them is, obviously, measuring transmission times in the different cases.
Several tests were performed as follows:

4.1.1 Test 1: Real and non real-time standard transmissions

The aim of this simple test was proving that a real-time connection and a non real-time connection without bandwidth overload spend the same time transmitting the same package at the same period.

Test conditions:

- No bandwidth overload
- Connections tested separately
- Same period and package size
- Same transmitted data

The absolute value of the transmission time was not the main interest because it has already been tested [7]. Instead of this, it was wanted to compare the magnitudes of the transmission.

Results:

The obtained result was that both real and non real-time spend the same time transmitting the same data, without overloads.
The difference between a real and non real-time transmission is the assigned bandwidth percentage. If the connections do not exceed the assigned bandwidth, as for the conditions of this experiment, there are not differences.
Thus, the result of the experiment is the expected.

4.1.2 Test 2: Overloaded non real-time transmissions

The aim of this test was to study the behaviour of an overloaded transmission and compare it to a non-overloaded one.

Before explaining the methodology of the test, some concepts concerning this experiment will be refreshed:
Non real-time traffic has a bandwidth percentage assigned. Normally, this percentage is lower than the real-time one, because the main purpose is to fulfil the real-time requirements. If the allocated percentage is low and the non real-time traffic exceeds its bandwidth, period parameters will be recalculated and the transmission might become slower. In Section 3.2.5 the way to deal with non real-time traffic is explained.

The maximum bandwidth of the connection is 100Mbit/s. The way to calculate the connection bandwidth is:

$$BW = (1/period\_us) \cdot 10^{6} \cdot nPackets \cdot (min\_frag\_data \cdot 8 + Header\ size);$$

Where,

$nPackets = size\ /\ min\_frag\_data$

$min\_frag\_data = 46\ bits$

$Header\ size = 528\ bits$

With the formulas provided above the bandwidth can be calculated. Now, the parameter (k) to calculate the bandwidth overload is defined:

$k = requested\ BW\ /\ free\ BW;$

If $k > 1 \rightarrow$ *overload exists*

If $k < 1 \rightarrow$ *overload does not exist*

When overload exists, the period is recalculated in order not to exceed the bandwidth limitation.
Three cases with three non real-time bandwidth percentages are studied in the experiment. Study cases are 1%, 10% and 50% of non real-time bandwidth. These

percentages were chosen to create an overload in the non real-time connection in some cases.

Test conditions:

- Non real-time transmissions
- Different packet sizes
- Overload exists intentionally
- Fixed period
- Three bandwidth percentage (1%, 10% and 50%)

The period is invariable for two reasons. Firstly, the experiment does not study the relation between period and time, for more information about the influence of the period over the transmission time, see [7]. Secondly, when overload exists, the period is recalculated, then, it is better to vary a non recalculated parameter as the packet size.

Results:

Table 4.2 shows the results of the overload parameter and transmission time for each packet size and for the three cases mentioned above.

Percentage:	1%		10%		50%	
Size (bytes)	Overload Parameter	Transmission Time (ms)	Overload Parameter	Transmission Time (ms)	Overload Parameter	Transmission Time (ms)
4000	77,952	79,67	7,796	9,33	1,55904	1,98
3500	68,992	69,50	6,8992	9,1	1,37984	1,67
3000	59,136	61,05	5,9136	7	1,18272	1,75
2500	49,28	50,50	4,928	6,33	0,9856	0,78
2000	39,424	40,58	3,9424	4,67	0,78848	0,83
1500	29,568	31,67	2,568	3,65	0,59136	0,78
1000	19,712	23,33	1,9712	2,8	0,39424	0,89
500	9,856	12,67	0,9856	1,28	0,19712	0,99
100	2,688	3,25	0,2688	0,93	0,05376	1,01
75	1,792	1,84	0,1792	1,15	0,03584	0,67
50	1,792	1,66	0,1792	1,33	0,03584	0,65

Table 4.2: Numeric test results

From the table, it is easy to see that the new period (recalculated when overload exists) is obtained multiplying the old period and the overload constant (k).

To have a better visualization of the results, Figure 4.2 shows the overload effect over the transmission time.

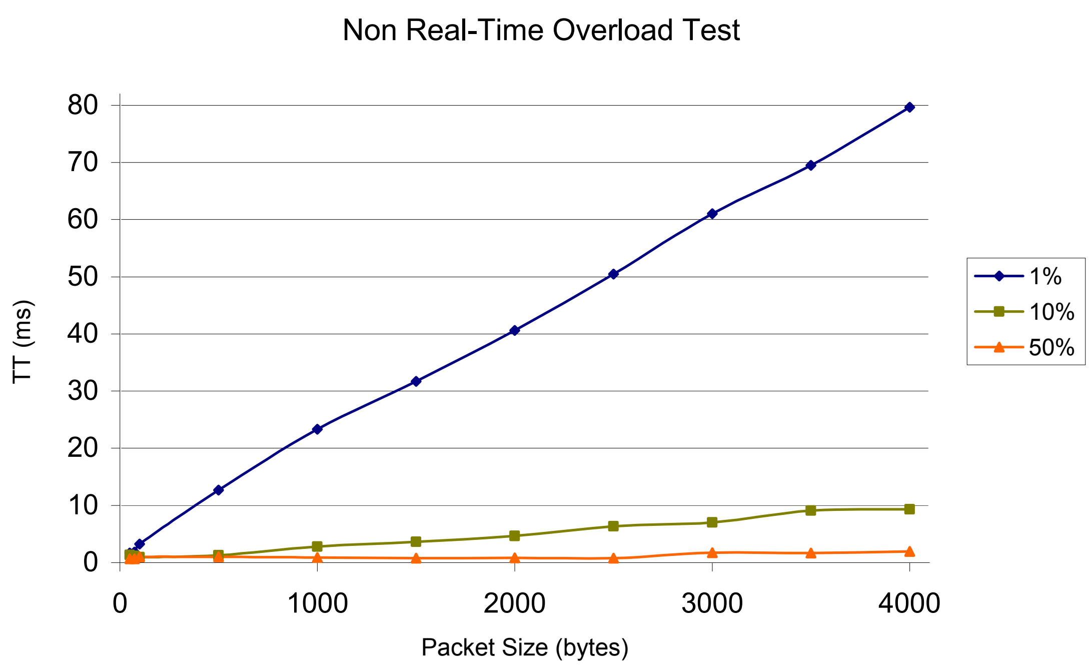


Figure 4.2: Graphical test result

The graphic shows clearly the influence of the overload over the transmission time. When the bandwidth percentage is lower (1%), the non real-time connection is overloaded if the packet size (size booked) is higher than 50 bytes. When the percentage is 10% the overloading shows up when the packet size is 500 bytes. With the 50% bandwidth 2500 bytes can be sent without overloading.
Therefore there is a proportional relation between packet size and bandwidth percentage given by the bandwidth equation written above.

The conclusion of this test is that an overload of the connection does increase the transmission time. Thus, setting the percentage of dedicated bandwidth to each transmission will be a key point.

4.1.3: Test 3: Simultaneous transmissions

The aim of this test was to study the case of two transmissions, one real-time and other non real-time, at the same time. The goal is to know the influence of sharing the same Ethernet card. The test launches two transmitter threads in one computer and two receiver threads in the other.

Test conditions:

- Not overload
- Fixed period and size reserved.
- Different size of transmitted packets.
- Transmissions at the same time.

The period and size reserved are fixed because the aim of the experiment is to observe the behaviour when the transmissions are at the same time.

Results:

The table 4.3 shows the time to transmit different packets in real and non real-time at the same time.

Transmission time(ms) Real Time	Transmission time (ms) Non Real Time	Size of the transmitted packet (bytes)
2,03	2,21	100
1,84	2,03	200
2,98	2,14	300
2,34	2,23	400
2,25	2,12	500

Table 4.3: Numeric test results

As it is appreciable in the table 4.3 when there is no overload the time does not depend on the size reserved
Looking at the table, the values are a bit higher than when these transmissions are independents. The reason of this fact is not clear. It can be consequence of the context change of the threads (sharing CPU) or due to the fact of sharing the same physical level.

4.2 Conclusions of the experiment

- The performance of ThrottleSim has been tested on a real platform.

- All the options implemented in the real-time environment (Chapter 3) have been tested. It was very useful to test the part related to the modifications of GlobeThrottle. Thus, the code was improved and some mistakes were found and corrected.

- The time-differences between real and non real-time have been studied. The results of the tests show that until the overload does not exist, both connections have the same performance. When there is overload (normally in the non real-time) because the allocated bandwidth percentage is not enough, the recalculation of the period makes the transmission slower.

5 Robot Experiments

The experimental platform of the thesis is introduced in this chapter, as well as the simulator where most of the experiments have been performed.
Experiments to integrate experimental and simulation tools with the real-time communications part and the trajectory generation for industrial robots are described. Results of the experiments are discussed as well.

5.1 Experimental platform

The experimental part of the thesis has been performed in the RobotLab (Department of Automatic Control, Lund Institute of Technology). In this Lab there are two ABB robots, the IRB2000 and the IRB6.

The robot used as experimental platform in this thesis is the *ABB Industrial Robot IRB6* (Figure 5.1). The original closed up system has been opened up in order to provide entire implementation control to the user. Also parts to create general interfaces were added. Mechanics, power electronics and safety systems were preserved.
The original ABB controller has been replaced by a VME-based computer connected to a Sun Workstation. References on the configuration [13]
The robot has five DOF. It means that given a desired position, the robot might not be able to reach it with also a desired orientation. It can lead to problems generating trajectories from random points because the user might not know if those points belong to the robot workspace. The robot has two cylindrical joints (one and five) and three revolute joints (two, three and four).

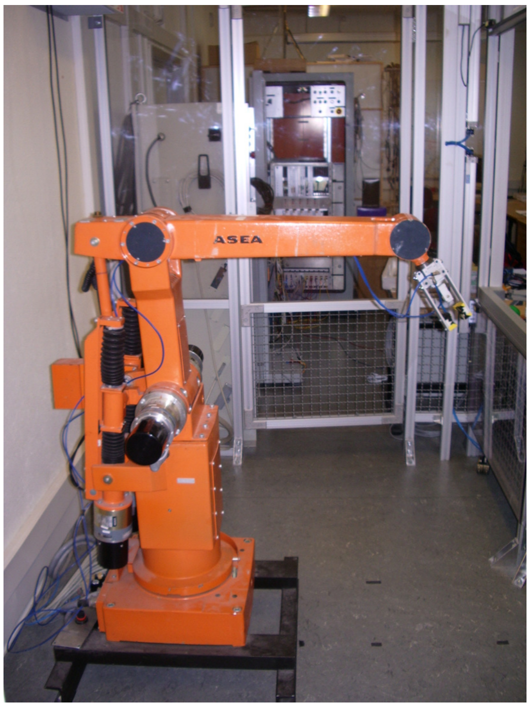


Figure 5.1: The ABB IRB6 robot

5.2 Simulation Platform [12]

The Java3D based simulation platform provides Java classes to represent the graphical visualisation and the kinematics of the robots. The different existing prototypes implement Java 3D robots with minimum rendering and navigation facilities created from a CAD geometry representation of the robot.
The goal of using this simulation platform is to have a portable 3D user interface where the movement of the robot can be simulated. Thus, we can work in userspace without compiling the Java code to C.

Kinematics of the robots is implemented accurately, but, at the moment, the dynamics provided in the package *roblet* are simple and the same for both robots.

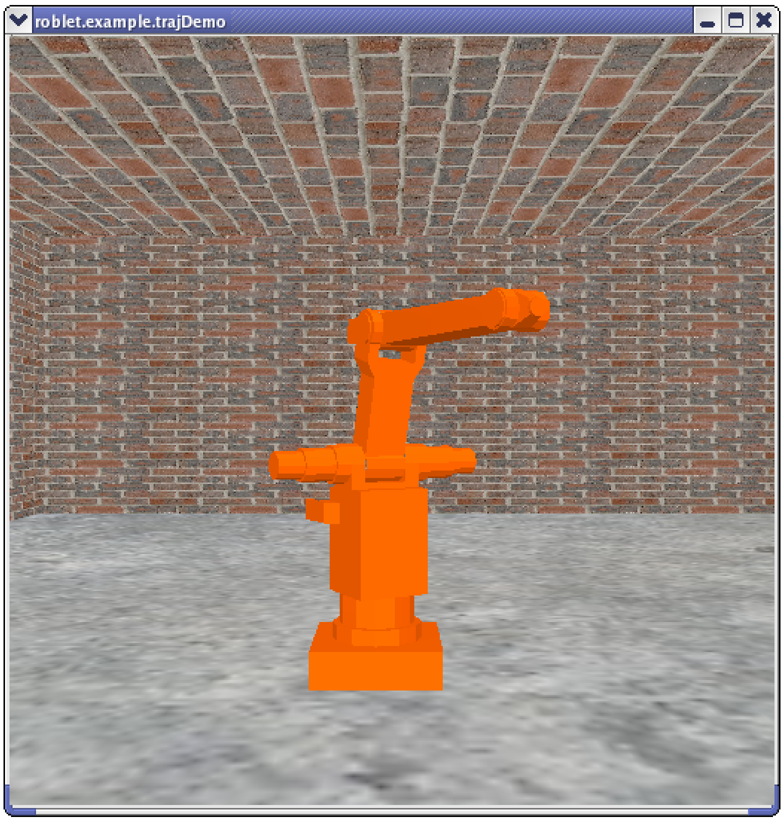


Figure 5.2: Graphical representation of the IRB6 robot

5.3 Experiments

So far, a real-time communications environment has been implemented and tested. Furthermore, a Java-based simulation platform and the IRB6 Robot have been presented.
In this chapter an application to integrate all the elements above-described will be implemented.

The aim of this application is to make the robot (or simulator) move following a trajectory generated with Matlab. The communication between the computer generating the trajectory and the robot (or simulator) should be carried out through ThrottleSim. To use ThrottleSim with the robot, all the packages described on the thesis (Chapter 3) and, also, ThrottleSim should be compiled using the Java2C compiler. Unfortunately, at this stage, it is not possible to compile all these packages. Therefore, in the first version of the demo, communications will be performed using the Java.net Socket class.

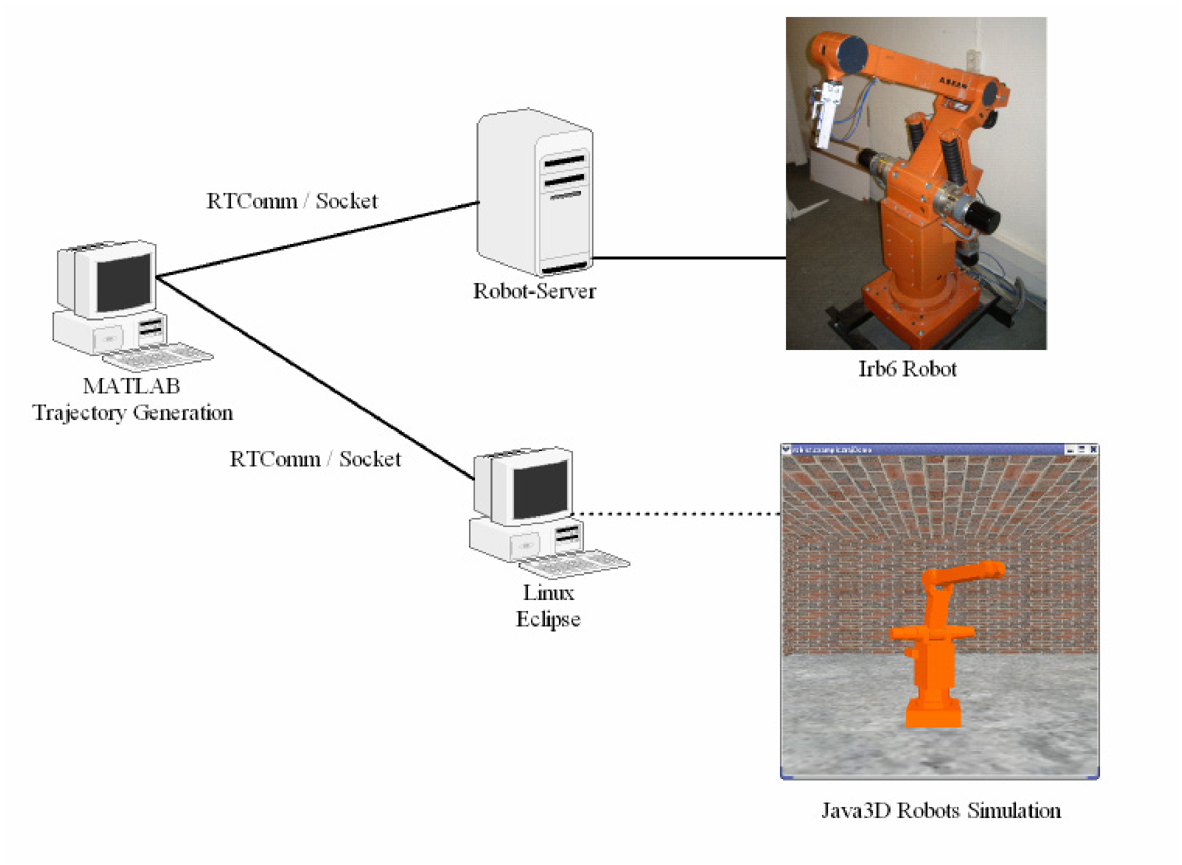


Figure 5.3: Experiment layout

Figure 5.3 shows the original design of the experiment, where the computer that generates the trajectory transmits data to the desired target.
All the stages of the experiment are described below:

5.3.1 Matlab trajectory generation [14] [15]

A trajectory is a path with a velocity profile and a time stamp. The path is the position reference for each time stamp. When a robot is moving between two points, a path is needed in order to control that all the joints are inside their working area during the movement. To have the complete control of the movement a velocity profile is requested as well. With these three elements the trajectory is generated.

To implement the path and trajectory generation, Matlab functions provided by the Path Generation Toolbox [16] will be used.

The first step to generate the trajectory is the path generation in Cartesian space. In the application implemented a three point linear motion is performed, then, a zone of radius z is defined around the intermediate point in order to make the path smooth. A fundamental requirement when constructing the zone path is that the resulting path should be continuous with continuous derivative. An example of a path is shown in Figure 5.5.

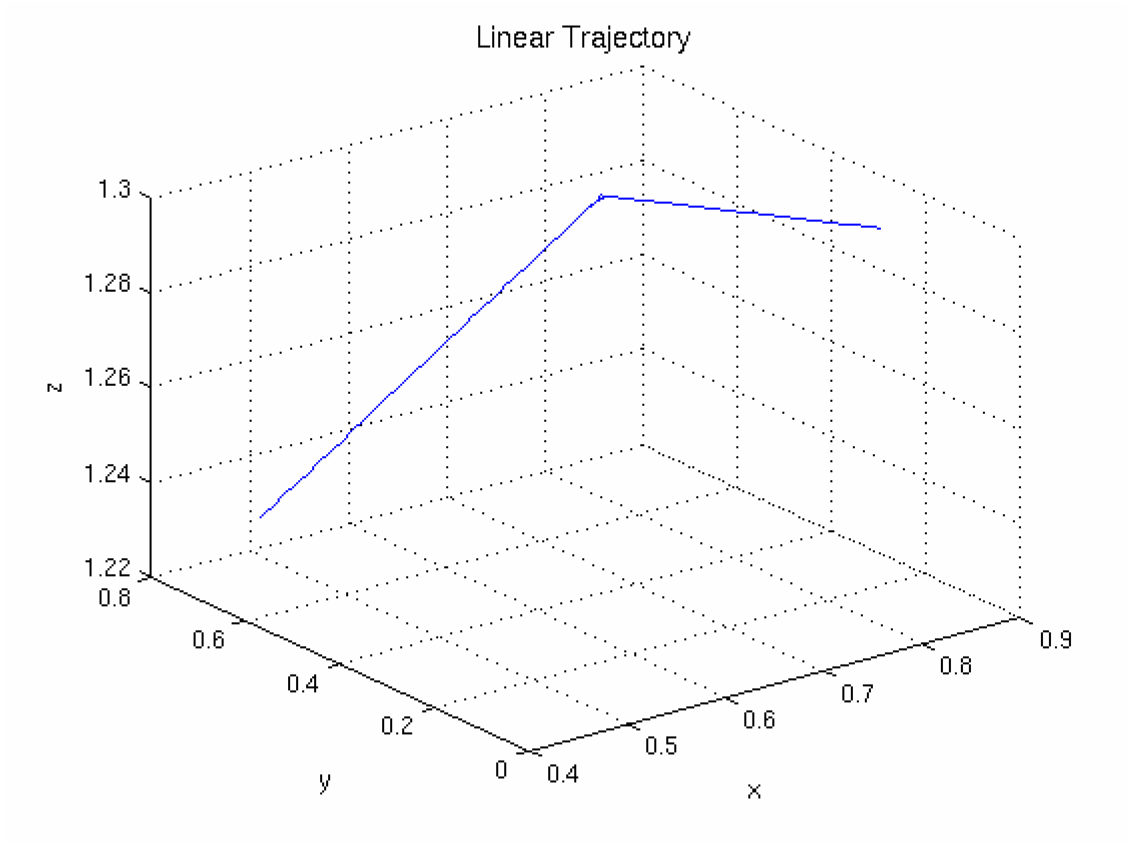


Figure 5.4: Example of a linear trajectory defined from three points

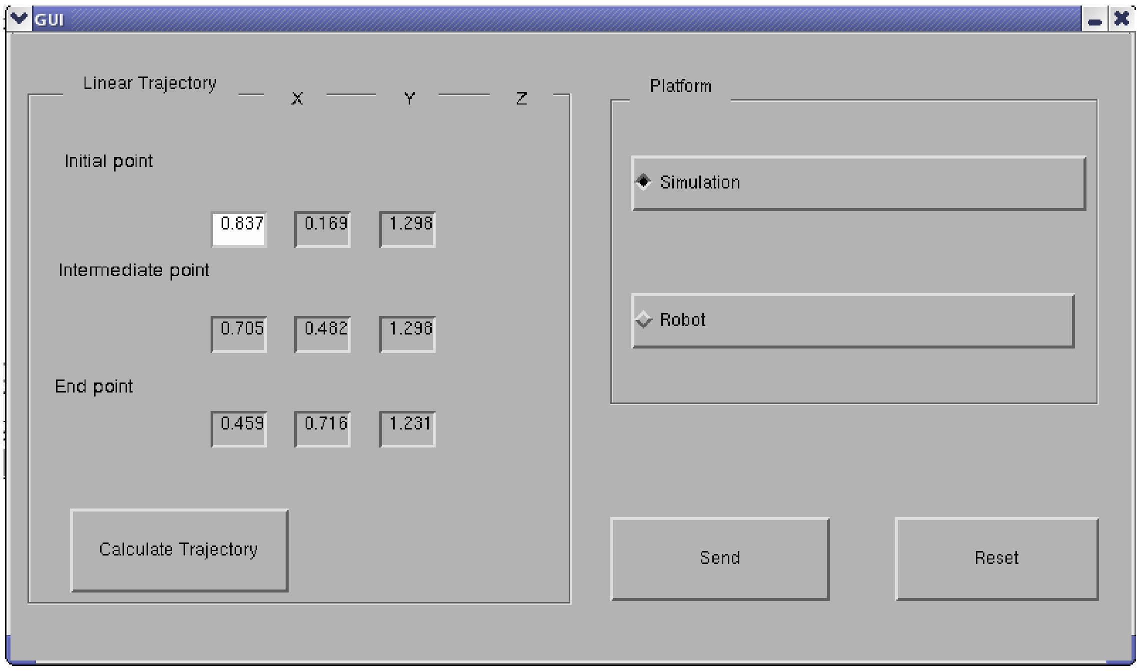
The second part of the calculation is the transformation from Cartesian path to Joint space using the kinematics model. The path in joint space is represented using cubic splines.
Once the path is generated, the next step is the generation of an optimized trajectory. To be optimum the path should be done in the minimum time and velocity and acceleration should remain within the specified limits.

As it is said before, the generation of the trajectory is done in Matlab and the variables obtained (position reference, velocity reference and time) are stored in MAT files.
Matlab functions provided by the Path Generation Toolbox give many possibilities to generate different trajectories for different robots (kinematics functions can be modified).
In this application only linear trajectories are performed because the aim of the experiment is not the complexity of the trajectory, but the fact of the robot following a suitable path.

The IRB6 robot has five degrees of freedom. It means that it is not able to reach all the points in the workspace with a certain orientation. The physical lengths of the robot links also limit the achievable workspace. That is something to consider in the generation of trajectories. If the selected points are out of the robot range, the trajectory will not be created properly.
Therefore, the points have to be chosen carefully in order to get a result.

5.3.2 Graphical User Interface (GUI)

As it is said in the last section (Matlab trajectory generation), a trajectory is defined with three points (start point , via point and end point). In the experiment these three points are chosen by the user through a Matlab Graphical User Interface [17] (Figure 5.5).



Linear Trajectory	X	Y	Z
Initial point	0.837	0.169	1.298
Intermediate point	0.705	0.482	1.298
End point	0.459	0.716	1.231

Platform: [✓ Simulation] Robot

Figure 5.5: Graphical User Interface

On the Linear Trajectory frame the user can choose the points (initial, intermediate and end point) that define the path to follow. After entering the points, the *Calculate Trajectory* button calls the Matlab file that generates a position reference, velocity reference and time vectors.
The frame platform gives the possibility for the user to choose the platform to run the application on. The user can choose between the Java3D simulation or real execution with the IRB6 Robot.
The Send button calls a Matlab file that loads the trajectory parameters and sends them to the robot server or the Java3D machine.
The Reset button sets all the parameters to their original values. The original values of the points are not zero because the IRB6 robot is not able to reach this position. Instead, other points that the robot can reach are chosen.

5.3.3 Matlab – Java interface [17]

A Matlab-Java interface is required in order to call Java from Matlab files. These Java calls can be referred either to a Java class created by us or to standard Java classes. The first version of the experiment has been performed using the java.net.Socket class instead of the RTComm classes, because the integration of the Socket to Matlab is easier. The implementation of the Matlab-Java interface for the packages RTComm and LabComm has not been possible due to the lack of time. Calls to the DataStructure were completed properly.

5.3.4 Control

Position control is performed for both the real robot system and for the simulator. The controller chosen has been a simple proportional (P) controller. This experiment is not focused on the control part, but, of course, it is necessary in order to have a stable behaviour.

5.3.5 Results

The Matlab trajectory generation has been tested for both robots in simulation with satisfactory results. But, this test could not achieve experimental results over the robot platform, because it was not possible to compile the Socket Class (java.lang.Socket). Neither the RTComm (ThrottleSim) has been compiled. Hence, communication between robots and the machine, where trajectory is generated, were not possible.
However, all the programs to transmit (sender and receiver sides) information to the robots are implemented in Java.
Also, a Graphical User Interface has been designed to support the trajectory generation and the communication for the simulator and the robots.
Finally, as it is said in the Matlab-Java interface section, Matlab calls to RTComm methods should be improved in the future. At this stage, only the DataStructure package can be called from Matlab.

6 Conclusions and Future Work

6.1 Summary

From the beginning, the thesis was dived into two parts: Java based real-time communications for robot systems and real-time control of industrial robots. These parts are strongly related. The projects started with the part concerning real-time communications.

The original goal of the thesis was to design and set up communication for a robot system using ThrottleNet (ThrottleSim). To achieve this objective, the implementation of a portable set of tools to support the real-time transmissions of control data was necessary. Since all the elements above-mentioned would be implemented in Java, the compilation of the entire communication environment, using the Java to C compiler, was another key point.

The starting point of the thesis was the study of the ThrottleNet-ThrottleSim network protocol [7] and the LabComm communication package [11] over TCP/UDP. To understand the work done so far and to implement a solution for the communication problem, the comprehension of the following concepts has been a key point:

- Real-time systems
- Real-time threading and synchronization
- Java in real-time
- Java serialization

The results of the implementations are a Java based packages laying over the ThrottleSim network protocol. These packages cover all the range of real and non real-time transmissions. And the sum of all of them is a real-time communication environment.

The second part of the thesis concerns the real-time control of the robots. The starting points of the second stage were the works related with real-time control in a Java platform and the experiences with trajectory generation for industrial robots.

The infrastructure for a further integration of robot control and communications in real-time is set up. Some problems related to Java to C compilation and Matlab-Java interfaces might be solved to make further progress in this field.

6.2 Future work

The topic of this Master thesis concerns different areas within the real-time systems. Furthermore, real-time systems have many aspects to investigate. Therefore, the future work is wide open.
There are two sources for the future work. The first one includes all the parts of this thesis not yet implemented because of lack of time or practical problems. The second source to suggest for future implementations are the ideas which came up during the

project. Normally these ideas are related to alternative implementations and improvements of the topics treated.
Firstly, the tasks that were considered as a goal or could have been considered as it during the development of the project will be described:

- Compile with the Java to C compiler all the packages implemented within the project, as well as ThrottleSim. Compilation of all these elements was tried during the project, but, at this stage, it is not possible to perform this translation-compilation. Several bugs were fixed, but there are still problems with the compiler concerning connections through sockets (java.net.Socket).

- Implement a Matlab-Java interface in order to transmit the Matlab generated trajectory using RTComm (ThrottleSim). This interface should be able to encapsulate the Matlab generated data in the Java-based data structure, establish the connection and transmit data using RTComm (ThrottleSim) methods.

- Improve the Java controller for the Irb6 robot. At this stage, the synchronization of the joints and the basic position control are done. But, more complex control modes could be performed. This software already exists, written in Modula-2, and should be ported to Java.

- Implement a dynamics class for the Java3D simulator in order to perform velocity control. Presently a very simple dynamics model is implemented; the same for both robots.

The proposals of future work explained below have come up during the development of the thesis:

- Improve the synchronization in the serialization process. The fact that the methods *writeObject* and *readObject* can not be controlled produces synchronization problems. These problems can be solved, but a delay in the process is the consequence.

- Design a method to dynamically book the percentage of bandwidth dedicated to the connection (real or non real-time). Presently this booking is performed statically through a constant of the GlobeThrottle class.

- Solve the packet loss problem of ThrottleNet and ThrottleSim when sending large amount of data.

6.3 Conclusions

From the beginning the thesis was meant to be more focused on the control of the industrial robots using compiled Java, fulfilling real-time demands. But, the time spent developing and testing the communication system and the practical problems of compilation changed the orientation of the thesis towards the real-time communications area. Consequently, the main achievements of the work are related to the real-time communications in Java. However the control part has been treated as well and a solid basis to develop real-time robot control applications has been set up.

The achievements of the thesis are the following:

A Java-based real-time communications environment. The most important features of the environment are the simplicity, the portability and the flexibility. Now, it is possible to perform a real-time transmission over ThrottleSim just calling a short number of simple methods. Also, it is possible to carry out real and non real-time transmissions separately or at the same time, allocating a percentage of the bandwidth to each transmission type.
A Java-based data structure can cover most of the message structures to be sent in a robot system context. And the implemented serialization mechanisms provide the possibility of sending large structures in the same transmission.

The infrastructure to set up the communications with the robots or simulator. Software to send and receive data over the ThrottleNet-ThrottleSim protocol has been implemented. These programs could not be tested on the real platform (robot) due to compiler problems.
Also, the synchronization and control structures for the ABB IRB6 robot have been improved.

Attempts to solve practical problems, especially in the robot experiments (Chapter 5) were done until the deadline of the project.

I hope this thesis will support a step ahead in the research in real-time communications and control carried out at Lund Institute of Technology (LTH).

Bibliography

[1] Juan Periset, *Portable Robot Programming*. Master's Thesis Report, Department of Computer Science. Lund Institute of Technology, Sweden, 2005

[2] Karl-Erik Årzén, *Real-Time Control Systems.* Lecture notes, 2004

[3] Anders Nilsson. Compiling Java for Real-Time Systems. Licentiate Thesis, Department of Computer Science, Lund Institute of Technology, Sweden, May 2004.

[4] Görel Hedin, Eva Magnusson. *JastAdd: an aspect-oriented compiler construction system.* Department of Computer Science, Lund University, Lund, Sweden

[5] Mark Roulo, *Java's Three Types of Portability*
http://www.javaworld.com

[6] Anders Blomdell, Karl-Erik Årzén, Anders Martinson. *ThrottleNet: Hard Real-Time Communication Using Switched Ethernet*. Department of Automatic Control.
Lund Institute of Technology, Sweden

[7] Daniel Nyberg, Patricia Grudziecka. *Real-Time Network Communication in Java*. . Master's Thesis Report, Department of Computer Science, Lund Institute of Technology, Sweden, 2004.

[8] Java platform website:
http://java.sun.com

[9] Elliote Rusty Harold. *Java I/O* O'Reilly & Associates, 1999.

[10] MatComm. Implemented by Anders Blomdell. Department of Automatic Control. Lund Institute of Technology, Sweden.

[11] Samuel Kasper. *Distributed Real Time Robot Vision in Java*. Master's Thesis Report, Department of Automatic Control. Lund Institute of Technology, Sweden. Winter 2003/2004

[12] Mathias Haage. *J3DSimulator: Flexible Interaction with Productive Robots in Partly Unstructured Environments.* Licentiete thesis, Department of Computer Science Lund Institute of Technology, 2004.

[13] Rolf Braun, Lars Nielsen and Klas Nilsson. *Reconfiguring an AESA IRB-6 Robot System for Control Experiments*. Department of Automatic Control. Lund Institute of Technology, October 1990

[14] M. Nyström, M. Norrlöf: *Path generation for industrial robots.*
Department of Electrical Engineering. Linköping University, Sweden, 2004

[15] Chin Yuan Chong. *Cooperative Robots.* . Master's Thesis Report,
Department of Automatic Control. Lund Institute of Technology, Sweden. March 2005

[16] M. Nyström, M. Norrlöf: PGT - *A path generation toolbox for Matlab(v0.1)*. Department of Electrical Engineering, Linköping University, Sweden, 2004

[17] Mathworks Homepage:
http://www.mathworks.com

Complementary Bibliography:

Johan Rix. *Deployment of embedded controllers on a Linux-based real-time Java platform* . Master's Thesis Report. Department of Computer Science. Lund Institute of Technology, Sweden, 2004.

Johan Rix. *Controlling IRB6 with Java Instructions paper*. Department of Automatic Control, Lund Institute of Technology, Sweden, 2004.

Eclipse Project Homepage
http://www.eclipse.org

Dept. of Automatic Control, LTH, Homepage:
http://www.control.lth.se

Dept. of Computer Science, LTH, Homepage:
http://www.cs.lth.se

LTH Homepage :
http://www.lth.se

Appendix: Implementations

A.1 DataStructure package

```
/*************************************/
 *      DataStructure.LabMessage      *
/************************************/

package DataStructure;

/**
 * This class is the superclass for LabObject and GeneralObject.
 * LabObject is the superclass for LabMatrix and LabStruct.
 * GeneralObject can be used as the superclass for general objects structs.
 * @version
 * @author
 */

public abstract class LabMessage implements Serializable {}

/*************************************/
 *      DataStructure.GeneralObject   *
/************************************/

package DataStructure;

/**
 * This class is the superclass for general objects structures.
 * @version
 * @author
 */

public abstract class GeneralObject extends LabMessage {}

/*************************************/
 *      DataStructure.LabObject       *
/************************************/

package DataStructure;

/**
 * This class is the superclass for LabMatrix and LabStruct.
 * A future version could also include arrays.
 * LabMatrix is the superclass for matrix classes of type double,
 * float, char, int and String.
 * LabStruct can be used as the superclass for custom structs.
 * @version
 * @author
 */

public abstract class LabObject extends LabMessage {
}

/*************************************/
 *      DataStructure.LabObject       *
/************************************/

package DataStructure;

/**
 * This class provides the fields and methods for a more general
 * datastructure than just matrices. Since it implements the
 * externalizable interface, it can be written to a stream and also
 * read from a stream.
 * This class is also the superclass for further even more specific
 * structs that may be used in combination with specific softwarepackages.
 * @version
 * @author
 */
public class LabStruct extends LabObject implements Externalizable {
    public static final String type_integer="int";
    public static final String type_float="float";
public static final String type_string="string";
public static final String type_char="char";
public  static final String type_double="double”;
private int range;
private UtilsExternal e;

/**
 *Mandatory no argument
 */
public LabStruct(){}

public LabStruct(int i){
   range=i;
}

Attribute[] field;
/**
 *Set the UtilsExternal param
 *@param e
 */
public void setUtils(UtilsExternal e){
   this.e=e;
}
public UtilsExternal getUtils(){
   return e;
}

/**
 * Gets the type of the position i .
 * @return The attribute`s type of the position i
 */
public String getType(int i) {
   return field[i].getType();
}

/*
 * Gets the value field of the position i.
 * @return The attribute`s value
 */
public LabObject getValue(int i) {
   return field[i].getValue();
}

/**
 * Sets the type and value of the position i.
 * @param type The type of the attribute
 * @param val The value of the attribute
 */
public void setAttribute(String type, LabObject val,int i) {
   if(field==null){
       field=new Attribute[range];
       for (int j=0;j<range;j++)
          field[j]=new Attribute();
   }
   field[i].SetAttribute(type,val);
}

/**
 * This function is used to save the state of the object. Since the class
 * implements Externalizable, this method must be implemented. Only the identity
 * of the class is automatically saved by the stream.
 * @serialData Write field-array field as object
 * @param out The ObjectOutput
 * @throws IOException
 */
public void writeExternal(ObjectOutput out) throws IOException {
   if(e!=null){
       e.writeOutInt(range);
   }
   else{
       out.writeInt(range);
   }
   for(int i=0;i<field.length;i++){
       field[i].writeExternal(out);
   }
}
```

```
    /**
     * This mandatory method reads in the data that was written out
     * in the writeExternal method. It restores its own fields. These fields
     * must be in the same order and type as they were written out.
     * @param in The ObjectInput
     * @throws IOException,ClassNotFoundException
     */
    public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException
{
       range=in.readInt();
       field=new Attribute[range];
       for (int j=0;j<range;j++)
           field[j]=new Attribute();
       for(int i=0;i<field.length;i++)
           field[i].readExternal(in);
    }

    /**
     *Give the range of the Attribute array
     */
    public int getRange(){
       return field.length;
    }
    private static final long serialVersionUID = 8563170928982865348L;

    class Attribute implements Externalizable{
       private   String type;
       private    LabObject value;

       public Attribute(){}

       /**
         * This function is used to save the state of the object. Since the class
         * implements Externalizable, this method must be implemented. Only the identity
         * of the class is automatically saved by the stream.
         * @param out The ObjectOutput
         * @throws IOException
         */
        public void writeExternal(ObjectOutput out) throws IOException {
           if(e!=null){
               e.writeOutObject(type);
               e.writeOutObject(value);
           }
           else{
               out.writeObject(type);
               out.writeObject(value);
           }
        }

        /**
         * This mandatory method reads in the data that was written out
         * in the writeExternal method. It restores its own fields. These fields
         * must be in the same order and type as they were written out.
         * @param in The ObjectInput
         * @throws IOException,ClassNotFoundException
         */
         public void readExternal(ObjectInput in) throws IOException,
       ClassNotFoundException {
               type=(String)in.readObject();
               value=(LabObject)in.readObject();
        }

       /**
        * @return the type
        */
       public String getType(){
           return type;
       }

       /**
        * @return the value
        */
       public LabObject getValue(){
           return value;
       }
```

```
        /**
         *@parameter name the type of the object
         * @parameter val the value of the object
         */
        public void SetAttribute(String name, LabObject val){
            type=name;
            value=val;
        }
    }
}

/*************************************/
 *DataStructure.LabMatrix.LabMatrix  *
/************************************/

package DataStructure.LabMatrix;

/**
 * This abstract class is the superclass for the following classes:
 * LabCharMatrix, LabFloatMatrix, LabDoubleMatrix,
 * LabIntMatrix and LabStringMatrix. It extends LabObject.
 * @version
 * @author
 */

public abstract class LabMatrix extends LabObject implements Serializable {
    public LabMatrix() {}

    public LabMatrix(int rows, int cols) {
        super();
        this.rows = rows;
        this.cols = cols;
    }

    /**
     * Allows access to the protected number of rows of the matrix
     * @return The number of rows
     */
    public int getRows() {
        return rows;
    }

    /**
     * Allows access to the protected number of cols of the matrix
     * @return The number of cols
     */
    public int getCols() {
        return cols;
    }

    /**
     * The print method prints a matrix to the standard output.
     */
    public void printMatrix() {}

    protected int rows = 0;
    protected int cols = 0;
}

/**************************************/
 *DataStructure.LabMatrix.LabCharMatrix*
/**************************************/
package DataStructure.LabMatrix;

/**
 * This class provides methods to handle the elements of a
 * 2 dimensional matrix consisting of char elements.
 * @version
 * @author
 */

public class LabCharMatrix extends LabMatrix implements Externalizable{
    /**
     * Mandatory no-arg constructor
     */
    public LabCharMatrix() {super();}
public LabCharMatrix(int rows, int cols) {
   super(rows,cols);
   matrix = new char[rows][cols];
}

/**
 * Allows access to all elements of the matrix.
 * @param row The row of the matrix
 * @param col The column of the matrix
 * @return The element with index row, col
 */
public char getElement(int row, int col) {
   return matrix[row][col];
}

/**
 * Allows to set the elements of the matrix.
 * @param row The row of the matrix
 * @param col The column of the matrix
 * @param value The value to be set
 */
public void setElement(int row, int col, char value) {
   matrix[row][col] = value;
}

/**
 * Prints the matrix to the standard output.
 */
public void printMatrix() {
   int i=0;
   for (int r=0; r<rows; r++) {
       for (int c=0; c<cols; c++) {
          i++;
          System.out.print(getElement(r,c)+"  ");
       }
       System.out.println();
   }
}

/**
 * Builds the matrix given a datavector.
 * @param data The datavector to be converted
 */
public void buildMatrix(char[] data) {
   int i=0;
   for (int r=0; r<rows; r++) {
       for (int c=0; c<cols; c++) {
          setElement(r,c,data[i]);
          i++;
       }
   }
}

/**
 * This function is used to save the state of the object. Since the class
 * implements Externalizable, this method must be implemented. Only the identity
 * of the class is automatically saved by the stream.
 * @serialData Write rows and cols field as integer and then write
 *             the matrix field as an object.
 * @param out The ObjectOutput
 * @throws IOException
 */
public void writeExternal(ObjectOutput out) throws IOException {
   if(e!=null){
       e.writeOutInt(rows);
       e.writeOutInt(cols);
       e.writeOutObject(matrix);
   }

   else{
       out.writeInt(rows);
       out.writeInt(cols);
       out.writeObject(matrix);
   }
}
```

```
    /**
     * This mandatory method reads in the data that was written out
     * in the writeExternal method. It restores its own fields. These fields
     * must be in the same order and type as they were written out.
     * @param in The ObjectInput
     * @throws IOException,ClassNotFoundException
     */
    public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException
{
        rows = (int)in.readInt();
        cols =(int)in.readInt();
        matrix = (char[][])in.readObject();
    }
    /**
     * Set the UtilsExternal
     * @param The UtilsExternal
     */
    public void setUtils(UtilsExternal e){
        this.e=e;
    }

    /**
     * get the UtilsExternal attribute e
     * @return e the UtilsExternal variable
     */
    public UtilsExternal getUtils(){
        return e;
    }

    private UtilsExternal e;

    private static final long serialVersionUID = -4846416413357133325L;
    private char[][] matrix;
}

/**************************************/
 *DataStructure.LabMatrix.LabDoubleMatrix*
/**************************************/
package DataStructure.LabMatrix;

/**
 * This class provides methods to handle the elements of a
 * 2 dimensional matrix consisting of double elements.
 * @version
 * @author
 */

public class LabDoubleMatrix extends LabMatrix implements Externalizable {
    /**
     * Mandatory no-arg constructor
     */
    public LabDoubleMatrix() {super();}

    public LabDoubleMatrix(int rows, int cols) {
        super(rows,cols);
        matrix = new double[rows][cols];
    }

    /**
     * Allows access to all elements of the matrix.
     * @param row The row of the matrix
     * @param col The column of the matrix
     * @return The element with index row, col
     */
    public double getElement(int row, int col) {
        return matrix[row][col];
    }

    /**
     * Allows to set the elements of the matrix.
     * @param row The row of the matrix
     * @param col The column of the matrix
     * @param value The value to be set
     */
    public void setElement(int row, int col, double value) {
        matrix[row][col] = value;
    }

    /**
     * Prints the matrix to the standard output.
     */
    public void printMatrix() {
       int i=0;
       for (int r=0; r<rows; r++) {
           for (int c=0; c<cols; c++) {
              i++;
              System.out.print(getElement(r,c)+"  ");
           }
           System.out.println();
       }
    }

    /**
     * Builds the matrix given a datavector.
     * @param data The datavector to be converted
     */
    public void buildMatrix(double[] data) {
       int i=0;
       for (int r=0; r<rows; r++) {
           for (int c=0; c<cols; c++) {
              setElement(r,c,data[i]);
              i++;
           }
       }
    }

    /**
     * This function is used to save the state of the object. Since the class
     * implements Externalizable, this method must be implemented. Only the identity
     * of the class is automatically saved by the stream.
     * @serialData Write rows and cols field as integer and then write
     *             the matrix field as an object.
     * @param out The ObjectOutput
     * @throws IOException
     */
    public void writeExternal(ObjectOutput out) throws IOException {
       if(e!=null){
           e.writeOutInt(rows);
           e.writeOutInt(cols);
           e.writeOutObject(matrix);
       }

       else{
           out.writeInt(rows);
           out.writeInt(cols);
           out.writeObject(matrix);
       }
    }

    /**
     * This mandatory method reads in the data that was written out
     * in the writeExternal method. It restores its own fields. These fields
     * must be in the same order and type as they were written out.
     * @param in The ObjectInput
     * @throws IOException,ClassNotFoundException
     */
    public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException
{
            rows = in.readInt();
            cols = in.readInt();
            matrix = (double[][])in.readObject();
    }

    /**
     * Set a UtilsExternal param
     * @param e
     */
    public void setUtils(UtilsExternal e){
       this.e=e;
    }
    /**
     * Get the UtilsExternal
     * @return e the UtilsExternal attribute
     */
    public UtilsExternal getUtils(){
       return e;
    }

    private UtilsExternal e;
    private static final long serialVersionUID = 3924761579548860791L;
    private double[][] matrix;
}
/*************************************/
 *DataStructure.LabMatrix.LabFloatMatrix *
/*************************************/
package DataStructure.LabMatrix;

/**
 * This class provides methods to handle the elements of a
 * 2 dimensional matrix consisting of float elements.
 * @version
 * @author
 */

public class LabFloatMatrix extends LabMatrix implements Externalizable {
    /**
     * Mandatory no-arg constructor
     */
    public LabFloatMatrix() {super();}

    public LabFloatMatrix(int rows, int cols) {
       super(rows,cols);
       matrix = new float[rows][cols];
    }

    /**
     * Allows access to all elements of the matrix.
     * @param row The row of the matrix
     * @param col The column of the matrix
     * @return The element with index row, col
     */
    public float getElement(int row, int col) {
       return matrix[row][col];
    }

    /**
     * Allows to set the elements of the matrix.
     * @param row The row of the matrix
     * @param col The column of the matrix
     * @param value The value to be set
     */
    public void setElement(int row, int col, float value) {
       matrix[row][col] = value;
    }

    /**
     * Prints the matrix to the standard output.
     */
    public void printMatrix() {
       int i=0;
       for (int r=0; r<rows; r++) {
           for (int c=0; c<cols; c++) {
              i++;
              System.out.print(getElement(r,c)+"  ");
           }
           System.out.println();
       }
    }

    /**
     * Builds the matrix given a datavector.
     * @param data The datavector to be converted
     */
    public void buildMatrix(float[] data) {
       int i=0;
       for (int r=0; r<rows; r++) {
            for (int c=0; c<cols; c++) {
               setElement(r,c,data[i]);
               i++;
            }
        }
    }

    /**
     * This function is used to save the state of the object. Since the class
     * implements Externalizable, this method must be implemented. Only the identity
     * of the class is automatically saved by the stream.
     * @serialData Write rows and cols field as integer and then write
     *             the matrix field as an object.
     * @param out The ObjectOutput
     * @throws IOException
     */
    public void writeExternal(ObjectOutput out) throws IOException {
       if(e!=null){
           e.writeOutInt(rows);
           e.writeOutInt(cols);
           e.writeOutObject(matrix);
       }
       else{
           out.writeInt(rows);
           out.writeInt(cols);
           out.writeObject(matrix);
       }
    }

    /**
     * This mandatory method reads in the data that was written out
     * in the writeExternal method. It restores its own fields. These fields
     * must be in the same order and type as they were written out.
     * @param in The ObjectInput
     * @throws IOException,ClassNotFoundException
     */
    public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException
{
        rows = in.readInt();
       cols = in.readInt();
       matrix = (float[][])in.readObject();
    }

    /**
     * Set a UtilsExternal param
     * @param e
     */
    public void setUtils(UtilsExternal e){
       this.e=e;
    }

    /**
     * Get the UtilsExternal
     * @return e the UtilsExternal attribute
     */
    public UtilsExternal getUtils(){
       return e;
    }
    private UtilsExternal e;
    private static final long serialVersionUID = 1480412143908926824L;
    private float[][] matrix;
}

/****************************************/
 *DataStructure.LabMatrix.LabIntMatrix *
/****************************************/
package DataStructure.LabMatrix;

/**
 * This class provides methods to handle the elements of a
 * 2 dimensional matrix consisting of integer elements.
 * @version
 * @author
 */
public class LabIntMatrix extends LabMatrix implements Externalizable {
```

```
/**
 * Mandatory no-arg constructor
 */
public LabIntMatrix() {super();}

public LabIntMatrix(int rows, int cols) {
   super(rows,cols);
   matrix = new int[rows][cols];
}

/**
 * Allows access to all elements of the matrix.
 * @param row The row of the matrix
 * @param col The column of the matrix
 * @return The element with index row, col
 */
public int getElement(int row, int col) {
   return matrix[row][col];
}

/**
 * Allows to set the elements of the matrix.
 * @param row The row of the matrix
 * @param col The column of the matrix
 * @param value The value to be set
 */
public void setElement(int row, int col, int value) {
   matrix[row][col] = value;
}

/**
 * Prints the matrix to the standard output.
 */
public void printMatrix() {
   int i=0;
   for (int r=0; r<rows; r++) {
       for (int c=0; c<cols; c++) {
          i++;
          System.out.print(getElement(r,c)+"  ");
       }
       System.out.println();
   }
}

/**
 * Builds the matrix given a datavector.
 * @param data The datavector to be converted
 */
public void buildMatrix(int[] data) {
   int i=0;
   for (int r=0; r<rows; r++) {
       for (int c=0; c<cols; c++) {
          setElement(r,c,data[i]);
          i++;
       }
   }
}

/**
 * This function is used to save the state of the object. Since the class
 * implements Externalizable, this method must be implemented. Only the identity
 * of the class is automatically saved by the stream.
 * @serialData Write rows and cols field as integer and then write
 *             the matrix field as an object.
 * @param out The ObjectOutput
 * @throws IOException
 */
public void writeExternal(ObjectOutput out) throws IOException {
   if(e!=null){
       e.writeOutInt(rows);
       e.writeOutInt(cols);
       e.writeOutObject(matrix);
   }
   else{
       out.writeInt(rows);
       out.writeInt(cols);
```

```
            out.writeObject(matrix);
        }
    }

    /**
     * This mandatory method reads in the data that was written out
     * in the writeExternal method. It restores its own fields. These fields
     * must be in the same order and type as they were written out.
     * @param in The ObjectInput
     * @throws IOException,ClassNotFoundException
     */
    public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException
{
         rows = in.readInt();
        cols = in.readInt();
        matrix = (int[][])in.readObject();
    }

    /**
     * Set a UtilsExternal param
     * @param e
     */
    public void setUtils(UtilsExternal e){
        this.e=e;
    }

    /**
     * Get the UtilsExternal
     * @return e the UtilsExternal attribute
     */
    public UtilsExternal getUtils(){
        return e;
    }
    private UtilsExternal e;
    private static final long serialVersionUID = -955999251175728919L;
    private int[][] matrix;
}
/*****************************************/
 *DataStructure.LabMatrix.LabStringMatrix *
/*****************************************/
package DataStructure.LabMatrix;

/**
 * This class provides methods to handle the elements of a
 * 2 dimensional matrix consisting of integer elements.
 * @version
 * @author
 */

public class LabStringMatrix extends LabMatrix implements Externalizable {
    /**
     * Mandatory no-arg constructor
     */
    public LabStringMatrix() {super();}

    public LabStringMatrix(int rows, int cols) {
        super(rows,cols);
        matrix = new String[rows][cols];
    }

    /**
     * Allows access to all elements of the matrix.
     * @param row The row of the matrix
     * @param col The column of the matrix
     * @return The element with index row, col
     */
    public String getElement(int row, int col) {
        return matrix[row][col];
    }

    /**
     * Allows to set the elements of the matrix.
     * @param row The row of the matrix
     * @param col The column of the matrix
     * @param value The value to be set
     */
    public void setElement(int row, int col, String value) {
        matrix[row][col] = value;
    }

    /**
     * Prints the matrix to the standard output.
     */
    public void printMatrix() {
        int i=0;
        for (int r=0; r<rows; r++) {
            for (int c=0; c<cols; c++) {
                i++;
                System.out.print(getElement(r,c)+"  ");
            }
            System.out.println();
        }
    }

    /**
     * Builds the matrix given a datavector.
     * @param data The datavector to be converted
     */
    public void buildMatrix(String[] data) {
        int i=0;
        for (int r=0; r<rows; r++) {
            for (int c=0; c<cols; c++) {
                setElement(r,c,data[i]);
                i++;
            }
        }
    }

    /**
     * This function is used to save the state of the object. Since the class
     * implements Externalizable, this method must be implemented. Only the identity
     * of the class is automatically saved by the stream.
     * @serialData Write rows and cols field as integer and then write
     * the matrix field as an object.
     * @param out The ObjectOutput
     * @throws IOException
     */
    public void writeExternal(ObjectOutput out) throws IOException {
        if(e!=null){
            e.writeOutInt(rows);
            e.writeOutInt(cols);
            e.writeOutObject(matrix);
        }
        else{
            out.writeInt(rows);
            out.writeInt(cols);
            out.writeObject(matrix);
        }
    }

    /**
     * This mandatory method reads in the data that was written out
     * in the writeExternal method. It restores its own fields. These fields
     * must be in the same order and type as they were written out.
     * @param in The ObjectInput
     * @throws IOException,ClassNotFoundException
     */
    public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException
{
         rows = in.readInt();
        cols = in.readInt();
        matrix = (String[][])in.readObject();
    }

    /**
     * Set a UtilsExternal param
     * @param e
     */
    public void setUtils(UtilsExternal e){
        this.e=e;
    }
```

```
   /**
    * Get the UtilsExternal
    * @return e the UtilsExternal attribute
    */
    public UtilsExternal getUtils(){
       return e;
    }
    private UtilsExternal e;
    private static final long serialVersionUID = -2112330479511158998L;
    private String[][] matrix;
}
```

A.2 RTComm package

```
/***************************/
 *     RTComm.Socket      *
/**************************/
package RTComm;

/**
 * The <code>Socket</code> class implements a socket used
 * to send objects through ThrottleNet. Socket is used in both, real and non-real time
transmissions.
 * @author
 * @version
 * @see ThrottleNetSocket
 */
abstract class Socket {
    /**
     * The utility needed for converting serialized objects to
     * arrays of bytes and vice versa.
     * utilsS is used to calculate the size of the largest object that can be sent
     * fulfilling real time restrictions.
     */
    protected UtilsSerial utilsS;

    /**
     * The utility needed for converting serialized objects to
     * arrays of bytes and vice versa.
     * utilsC is used to send and receive serializable objects.
     */

    protected UtilsSerial utilsC;

    /**
     * The name of the connection between two nodes.
     */
    protected String name;

    /**
     * The amount of data to be sent in bytes.
     */
    protected int size;

    /**
     * The object whose size it will the minimum that it can be send it.
     */
    protected Object besteffort;

    /**
     * The time between two packets in microseconds.
     *
     */
    protected long period_us;

    /**
     * Constructs a Socket.
     * @param   name      the name of the connection.
     * @param   period_us the time between two packets in microseconds.
     */
    public Socket(String name,int period_us)throws IOException {
        this.name = name;
        utilsS=new UtilsSerial();
        utilsC=new UtilsSerial();
        this.period_us = period_us;
    }

    /**
     * Constructs a Socket.
     * @param   name      the name of the connection.
     * @param besteffort the smaller object which can be send.
     * @param   period_us the time between two packets in microseconds.
     */

public Socket(String name, Object besteffort, int period_us)throws IOException{
        this.name = name;
        utilsS=new UtilsSerial();
        this.size=utilsS.getObjectSize((Serializable) besteffort);
        this.period_us = period_us;
    }
}

/***************************/
 *     RTComm.RTSocke     *
/*************************/

package RTComm;

/**
 * The <code>RTSocket</code> class implements a socket used
 * to send objects through ThrottleNet, fulfilling temporal restrictions.
 * @author
 * @version
 * @see ThrottleNetSocket
 */
abstract class RTSocket extends Socket{

    /**
     * The utility needed for converting externalized objects to
     * arrays of bytes and vice versa.
     */
    protected UtilsExternal utilsE;

    /**
     * The amount of data to be sent in bytes. Calculated from the largest object that
can
     * be sent fulfilling real time restrictions.
     *
     */
    protected int size

        /**
         * Constructs a RTsocket.
         * @param   name      the name of the connection.
         * @param   objsize   the larget object (LabObject) that can be sent fulfilling
temporal restrictions.
         * @param   period_us the time between two packets in microseconds.
         *
         */
        public RTSocket(String name, LabObject objsize,int period_us)
        throws IOException {
        super(name,period_us);
        utilsE = new UtilsExternal();
        this.size=utilsS.getObjectSize(objsize);
    }
}

/***************************/
 *     RTComm.RTTransmit *
/*************************/

package RTComm;

/**
* The <code>RTTransmit</code> class describes an intermediate layer, between the
application and ThrottleNet, for
* transmiting LabObject. The class implements methods to connect to GlobeThrottle and
* transmit the externalized or serialized data structure.
* It uses an external package, LabComm, to avoid the size limitation of ObjectInput and
ObjectOutput.
* @author
*/

public class RTTransmit extends RTSocket{
    private int dead_line;
    private byte[] b;
    private LabStruct labstr;
    private ThrottleNetTransmitSocket tts;



    /**
     * Constructs and initializes a  RTTransmit .
     * @param   name    the name of the connection.
     * @param   size    the largest object (LabObject) that can be sent to fulfill the
real time restrictions.
     * @param   period_us the time between two packets in microseconds.
     * @exception IOException if an I/O error occurs.
     */
    public RTTransmit(String name, LabObject size,int period_us)throws IOException {
       this(name,size, period_us,Integer.MAX_VALUE);
    }

    /**
     * Constructs and initializes a  RTTransmit .
     * @param   name      the name of the connection.
     * @param   size    the largest object (LabObject) that can be sent to fulfill the
real time restrictions.
     * @param   period_us the time between two packets in microseconds.
     * @param   dead_line time to deliver each packet in microseconds.
     * @exception IOException if an I/O error occurs.
     */
    public RTTransmit(String name, LabObject size,int period_us,int dead_line) throws
IOException{
       super(name,size,period_us);
       this.dead_line=dead_line;
       this.tts= new ThrottleNetTransmitSocket();
    }

    /**
     * Opens a connection with the specified properites.
     * @exception IOException     if an I/O error occurs.
     */
    public void connect ()throws IOException{
       // connects to GlobeThrottle, if the connection is not available, waits and tries
       again.
       tts.connect(name,size,(int)period_us,Utils.type_realtime);
    }

    /**
     * Transmits an externalized object over ThrottleNet.
     * @param exter The externalizable object which is transmited
     */

    public void transmit(Externalizable exter){
       //Reads from the PipedInputStream and save all the information in a byte array,
       //At the same time  PipedOutputStream is filled up using externalize method.
       try{
           ReadThread rt;
           UtilsExternal utilsE=new UtilsExternal();
           rt=new ReadThread(utilsE);
           LabStruct l=(LabStruct)exter;
           l.setUtils(utilsE);
           rt.start();
           utilsE.externalize(l);
           l.getUtils().automaticwakeup();
           // Writes in the byte[] b the array which has been created in ReadThread
           boolean data =true;
           while(data){
              if(utilsE.available()==0)
                  data=false;
           }
           byte[]b1=utilsE.read();
           //Transmits the byte array with all the data
```

```
            tts.transmit(b1);
        }catch(IOException e){System.out.println(e);}
        catch(InterruptedException ei){System.out.println(ei);}
    }

    /**
     * Transmits a Serializable object over ThrottleNet.
     * @param   serial the Serializable object which is transmited
     */
    public void transmit(Serializable serial){
        //Reads from the PipedInputStream and save all the information in a byte array,
        //At the same time  PipedOutputStream is filled up using convert method.
        try{
            ReadThreadS rts;
            UtilsSerial utilsC=new UtilsSerial();
            rts=new ReadThreadS(utilsC);
            rts.start();
            utilsC.convert(serial);
            // When there is not more data to be read, the ReadThreadS thread is
            interrupted
            boolean data=true;
            while(data){
                if(utilsC.available()!=0)
                    data=true;
                else{
                    data=false;
                    rts.interrupt();
                }
            }
            // Writes in the byte[] b the array which has been created in ReadThreadS
            byte[] b=utilsC.read();
            //Transmits the byte array with all the data
            tts.transmit(b);
        }catch(IOException e){System.out.println(e);}
        catch(InterruptedException ei){System.out.println(ei);}
    }

    /**
     * Closes the connection
     *
     */
    public void close(){
        tts.close();
    }
}

/***************************/
 *     RTComm.RTReceive *
/*************************/

package RTComm;

/**
 * The <code>RTReceive</code> class describes an intermediate layer, between the
 *application and ThrottleNet, for receiving objects. The class implements methods to
 *register the ThrottleNet socket and receive the information.
 * It uses an external package, LabComm, to avoid the size limitation of ObjectInput and
 *ObjectOutput.
 * @author  Juan Periset & Ferran Carlas
 */
public class RTReceive extends RTSocket{
    private int dead_line;
    private ThrottleNetReceiveSocket trs;
    private byte[] b=new byte[UtilsExternal.BUFFERSIZE];

    /**
     * Constructs and initializes a RTReceive .
     * @param   name   the name of the connection.
     * @param   size   the larget object(LabObject)that can be sent fulfilling temporal
restrictions.
     * @param   period_usthe time between two packets in microseconds.
     * @exception IOException     if an I/O error occurs.
     */
    public RTReceive(String name, LabObject size ,int period_us)throws IOException {
        this(name, size,period_us,Integer.MAX_VALUE);
    }
    /**
     * Constructs and initializes a  RTReceive .
     * @param   name     the name of the connection.
     * @param   size     the larget object (LabObject) that can be sent fulfilling
temporal restrictions.
     * @param   period_us  the time between two packets in microseconds.
     * @param   dead_line  time to deliver each packet in microseconds.
     * @exception IOException if an I/O error occurs.
     */
    public RTReceive(String name, LabObject size,int period_us,int dead_line) throws
IOException{
       super(name,size,period_us);
       this.dead_line=dead_line;
       this.trs=new ThrottleNetReceiveSocket();
    }

    /**
     * Registers the ThrottleNet receive socket.
     * @exception IOException     if an I/O error occurs.
     */
    public void register()throws IOException {
       //Registers the connection
       trs.register(name,size,(int)period_us,dead_line,Utils.type_realtime);
    }

    /**
     * Receives an externalizable object as byte array.
     * @exception IOException     if an I/O error occurs.
     * @exception PacketLossException if a packet is lost.
     * @exception ClassNotFoundException if no definition for the class could be
     *            found.
     * @return LabStruct received
     */
    public Externalizable receive(Externalizable object)throws
PacketLossException,IOException,ClassNotFoundException{        // Receives data in a
byte array
       b=trs.receive();
       WriteThread wt;
       UtilsExternal utilsE=new UtilsExternal();
       wt=new WriteThread(b,utilsE);
       wt.start();
       utilsE.externalizeRead(object);
       //Interrupts WriteThread when there is no more data available
       boolean data=true;
       while(data){
           if(utilsE.available()!=0)
              data=true;
           else{
              data=false;
              wt.interrupt();
           }
       }
       return object;
    }

    /**
     * Receives a Serializable object reconstructed from a byte array.
     * @exception IOException     if an I/O error occurs.
     * @exception PacketLossException if a packet is lost.
     * @exception ClassNotFoundException if no definition for the class could be
     *            found.
     * @return Serializable received
     */
    public Serializable receive()throws PacketLossException,
IOException,ClassNotFoundException{
       // Receives data in a byte array
       Serializable obj=null;
       b=trs.receive();
       //Closes the ThrottleNetReceiveSocket
       trs.close();
       //Writes the buffer using utils and externallizes at the same time this buffer
       // storeing the buffer in a serializable object.
       WriteThreadS wts;
       UtilsSerial utilsC=new UtilsSerial();
       wts=new WriteThreadS(b,utilsC);
       wts.start();
       obj=utilsC.convertS();
```

```
        //Interrupts WriteThreadS when there is no more data available
        boolean data=true;
        while(data){
            if(utilsC.available()!=0)
               data=true;
            else{
               data=false;
               wts.interrupt();
            }
        }
        return obj;
    }


    /**
     * Close the connection
     */
    public void close(){
        trs.close();
    }
}

/**************************/
 *     RTComm.Transit     *
/*************************/
package RTComm;

/**
* The <code>Transmit</code> class describes an intermediate layer, between the
application and ThrottleNet, for
* transmiting data in not real time. The class implements methods to connect to
GlobeThrottle and
* transmit the date.
* It uses an Labcomm serial package, LabComm.Serial, to avoid the size limitation of
ObjectInput and ObjectOutput.
* @author
*/

public class Transmit extends Socket{
        private int dead_line;
        private byte[] b;
        private LabStruct labstr;
        private ThrottleNetTransmitSocket tts;

    /**
     * Constructs and initializes a  Transmit .
     * @exception IOException     if an I/O error occurs.
     */

    public Transmit(String name, Object besteffort,int period_us)throws IOException {
        this(name, besteffort, period_us, Integer.MAX_VALUE);
    }

    /**
     * Constructs and initializes a  Transmit .
     * @param   name      the name of the connection.
     * @param   size      the amount of data in the ThrottleNet packet.
     * @param   period_us   the time between two packets in microseconds.
     * @param   dead_line   time to deliver each packet in microseconds.
     * @exception IOException if an I/O error occurs.
     */
    public Transmit(String name, Object besteffort,int period_us,int dead_line) throws
IOException{
        super(name,besteffort, period_us);
        this.dead_line=dead_line;
        this.tts= new ThrottleNetTransmitSocket();
    }
    /**
     * Opens a connection with the specified properites.
     * @exception IOException     if an I/O error occurs.
     */
    public void connect ()throws IOException{
        //connects to GlobeThrottle, if the connection is not available, waits and tries
        again.
        tts.connect(name,size,(int)period_us,Utils.type_nrealtime);
    }
    /**
     * Transmits a serializable object over ThrottleNet.
     *@param serial Serializable object
     */
    public void transmit(Serializable serial){
       try{
           ReadThreadS rts;
           UtilsSerial utilsC=new UtilsSerial();
           rts=new ReadThreadS(utilsC);
           rts.start();
           utilsC.convert(serial);
           // When there is not more data to be read, the ReadThreadS is interrupted
           boolean data=true;
           while(data){
              if(utilsC.available()!=0)
                  data=true;
              else{
                  data=false;
                  rts.interrupt();
                  rts.join();
              }
           }
           // Writes in the byte[] b the array which has been created in ReadThreadS
           b=utilsC.read();
           tts.transmit(b);

       }catch(IOException e){System.out.println(e);}
       catch(InterruptedException ei){ System.out.println(ei);}
    }

    /**
     * Closes the connection
     *
     */
    public void close(){
       tts.close();
    }
}

/**************************/
 *     RTComm.Receive *
/*************************/
package RTComm;

/**
 * The <code>Receive</code> class describes an intermediate layer, between the
 *application and ThrottleNet, for receiving serializable objects.The class implements *
 *methods to register
 * the ThrottleNet socket and receive the information.
 * It uses an external package, LabComm, to avoid the size limitation of ObjectInput and
 * ObjectOutput
 * @author
 */
public class Receive extends Socket{
    private int dead_line;
    private ThrottleNetReceiveSocket trs;
    private byte[] b=new byte[UtilsSerial.BUFFERSIZE];

    /**
     * Constructs and initializes a Receive .
     * @exception IOException if an I/O error occurs.
     */
    public Receive(String name,Object besteffort,int period_us)throws IOException {
       this(name, besteffort, period_us, Integer.MAX_VALUE);
    }

    /**
     * Constructs and initializes a  Receive .
     * @param   name     the name of the connection.
     * @param   size     the amount of data to be received in the packet.
     * @param   period_us  the time between two packets in microseconds.
     * @param   dead_line  time to deliver each packet in microseconds.
     * @exception IOException if an I/O error occurs.
     */
       public Receive(String name,Object besteffort,int period_us,int dead_line) throws
IOException{
       super(name,besteffort, period_us);
```

```
        this.dead_line=dead_line;
        this.trs=new ThrottleNetReceiveSocket();
    }

    /**
     * Registers the ThrottleNet receive socket.
     * @exception IOException if an I/O error occurs.
     */
    public void register()throws IOException {
        //Registers the connection
        trs.register(name,size,(int)period_us,dead_line,Utils.type_nrealtime);
    }


    /**
     * Receives an serializable object reconstructed from a byte array.
     * @exception IOException     if an I/O error occurs.
     * @exception PacketLossException if a packet is lost.
     * @exception ClassNotFoundException if no definition for the class could be
     *           found.
     * @return serializable object received
     */
        public Serializable receive()throws PacketLossException,
IOException,ClassNotFoundException{
        b=trs.receive();
        WriteThreadS wts;
        UtilsSerial utilsC=new UtilsSerial();
        wts=new WriteThreadS(b,utilsC);
        wts.start();
        Serializable object=utilsC.convertS();
        //Interrupts WriteThreadS when there is no more data available
        boolean data=true;
        while(data){
            if(utilsC.available()!=0)
               data=true;
            else{
               data=false;
               wts.interrupt();
            }
        }
        return object;
    }

    /**
     * Close the connection
     */
    public void close(){
        trs.close();
    }
}
```

A.3 LabComm Package

```
/**********************************/
 * LabComm.External.UtilsExternal *
/**********************************/
package LabComm.External;

/**
 * The <code>UtilsExternal</code> class is a util class that contains methods
 * for converting externalized objects to arrays of bytes
 * and vice versa.
 * It works using the interface Externalizable and let every externalizable object to
convert its content
 * into array of bytes. While reading, the object is reconstructed from this content.
 * @author
 */
public final class UtilsExternal {
     /**
     * Private buffer to store data.
     */
     private byte[] b=new byte[BUFFERSIZE];
    /**
     *Boolean used to syncronize methodes write and read of this class.
     */
    private boolean availableRead;
    /**
     *Boolean used to syncronize methodes writeOut and readPin.
     */
    private boolean wakeup=false;
    /**
     *Boolean used to indicate the end of the externalization process.
     */
    private boolean exterover=false;
    /**
     * int used to store the number of bytes readed in the readPin method.
     */
    private int length=0;
    /**
     * Piped output stream used for conversion.
     */
    private PipedOutputStream pout;
    /**
     * Piped input stream used for conversion.
     */
    private PipedInputStream pin;
    /**
     * Object output stream used for conversion of objects to byte arrays.
     */
    private ObjectOutputStream out;
    /**
     * Object input stream used for conversion of byte arrays to objects.
     */
    private ObjectInputStream in;
    /**
     * Size of the PipedBuffer
     */
    static final int  PIPESIZE=1024;
    /**
     *Sized of the auxiliar buffer it can be modified if it is necessary
     */
    public static final int  BUFFERSIZE=128000;
    /**
     * Constructs an utility object and initializes all streams.
     *
     * @exception IOException if an I/O error occurs.
     */
    public UtilsExternal() throws IOException {
       initialize()
    }



    /**
     * Initializes the streams.
     * @exception IOException if an I/O error occurs.
     */
    private void initialize() throws IOException {
       pout = new PipedOutputStream();
       pin = new PipedInputStream(pout);
       out = new ObjectOutputStream(pout);
       out.flush();
       in = new ObjectInputStream(pin);
     }

    /**
     * Externalizes an object.It writes the object in a PipedOutputStream.
     * The object must implement externalizable.
     * @exception IOException if an I/O error occurs.
     */
    public void externalize(Externalizable ext) throws IOException {
       out.reset();
       ext.writeExternal(out);
       out.flush();
    }

    /**
     * Externalizes the object.It reads the data from the PipedInputStream and converts
     * it to an object with the structure it had before. The object must implement
     * externalizable.
     * @exception IOException if an I/O error occurs and ClassNotFoundException when
     * the class does't match with the class of the expected data.
     */
    public void externalizeRead(Externalizable obj)
       throws IOException, ClassNotFoundException {
       pout.flush();
       obj.readExternal(in);
    }

    /**
     * Writes the parameter buff ina private buffer.
     * @param buff The buffer from whom we take the information.
     */
    public synchronized void write( byte[] buff){
       b=buff;
    }

    /**
     * Writes length number of bytes of the buffer starting from offset in
     * PipedOutputStream.
     * @param buff It is the buffer from whom we take the information.
     * @param offset
     * @param length
     * @exception IOException if an I/O error occurs.
     */
    public synchronized int  writePout( byte[] buff,int offset,int length){
       try{
           pout.write(buff,offset,length);

       }catch(IOException e){System.out.println("Inside utils writePout"+e);}
       return length;
    }

    /**
     * Returns private buffer b.
     * @return a byte[].
     */
    public synchronized byte[] read(){
       byte[] buff=new byte[length];
       for(int i=0;i<length;i++)
           buff[i]=b[i];
       return buff;
    }

    /**
     * Reads from PipedInputStream and stores the bytes in the buffer b.
     * @return the number of bytes read.
     * @exception IOException if an I/O error occurs.
     */
    public synchronized int readPin(int offset){
       int bytesread=-1;
       try {
           while(!wakeup){
              try{
                  wait();
              }catch(InterruptedException e1){e1.printStackTrace();}
           }
           bytesread=pin.read(b,offset,available());
           offset+=bytesread;
           length=offset;
           wakeup=false;
           notifyAll();
       }catch(IOException e){System.out.println("Inside utils readPin"+e);}
       return offset;
    }

    /**
     * Returns the number of bytes that can be read from PipedInputStream.
     * @return the number of bytes that can be read from PipedInputStream.
     * @exception IOException if an I/O error occurs.
     */
    public int available(){
       int bytes=-1;
       try{
           bytes=pin.available();
```

```
    }catch(IOException e){System.out.println(e);}
    return bytes;
}

/**
 * Wakes up the readPin when the extarnalization process over
 */
public synchronized void automaticwakeup(){
    exterover=true;
    wakeup=true;
    notifyAll();
}

/**
 * Checks if the externalization process has ended
 */
public synchronized boolean IsOver(){
    return exterover;
}

/**
 * Writes an integer in the ObjectOutpuStream and notifies it.
 *@param value
 */
public synchronized void writeOutInt(int value){
    try{
        while(pin.available()>4){
            try{
                wakeup=true;
                notifyAll();
                wait();
            }catch(InterruptedException e1){e1.printStackTrace();}
        }
        out.writeInt(value);
        notifyAll();
    }catch(IOException e){e.printStackTrace();}
    wakeup=true;
    notifyAll();
}

/**
 * Writes an object in ObjectOutputStream and notifies it.
 *@param value
 */
public synchronized void writeOutObject(Object value){
    try{
        while(pin.available()>900){
            try{
                wakeup=true;
                notifyAll();
                wait();
            }catch(InterruptedException e1){e1.printStackTrace();}
        }
        out.writeObject(value);
        notifyAll();
    }catch(IOException e){e.printStackTrace();}
    wakeup=true;
    notifyAll();
}

/**
 *Returns the size of the object after externalizing it.
 *@param exter
 *@return
 */
public int getObjectSize(Externalizable exter) {
    byte[] b1=null;
    try{
        ReadThread rt;
        UtilsExternal utilsE=new UtilsExternal();
        rt=new ReadThread(utilsE);
        LabStruct l=(LabStruct)exter;
        l.setUtils(utilsE);
        rt.start();
        utilsE.externalize(l);
        l.getUtils().automaticwakeup();
        boolean data =true;
```

```
            while(data){
                if(utilsE.available()==0)
                    data=false;
            }
            b1=utilsE.read();
        }catch(IOException e){e.printStackTrace();}
        return b1.length;
    }
}

/**********************************/
* LabComm.External.ReadThread      *
/*********************************/

package LabComm.External;

/**
 *This thread reads from the PipedInputStream and save all the information in a byte
array,
 *All data is stored in a private buffer utils.java,
 *TransmiterThread will read this buffer and it will send its content.
 *@author
 */

public class ReadThread extends RTThread{
    private byte[] b=new byte [UtilsExternal.BUFFERSIZE];
    private UtilsExternal u;
    /**
     * Constructor
     * @param ut
     */
    public ReadThread(UtilsExternal ut){
       u = ut;
    }

    public void run(){
       int offset=0;
       int bytesread;
       // Read while there the externalization process does not finish or there ara data
       while(!u.IsOver()|| u.available()!=0){
           offset=u.readPin(offset);
       }
    }
}
```

```
/**********************************/
 * LabComm.External.WriteThread   *
/*********************************/
package LabComm.External;

/**
 * This thread writes the received byte array in a PipedOutputStream.
 * The information is written in blocks of 1024bytes, except for the last block that has
the bytes left.
 * @author
 */
public class WriteThread extends RTThread{
       byte[] bout=new byte[UtilsExternal.BUFFERSIZE];
       UtilsExternal u;
    /**
     *Constructor
     *@param b byte buffer
     *@param ut UtilsExternal
     */
    public WriteThread(byte[] b,UtilsExternal ut){
       bout=ab;
       u=ut;a
    }a
    puablic void run(){
       int offset=0;
```

```
        int count=0;
        //it is interrupted by the receiver thread when this finishes externalize method.
        while(!isInterrupted()){
            if(offset<bout.length){
               if((bout.length-offset)>=UtilsExternal.PIPESIZE){
                   count=u.writePout(bout,offset,UtilsExternal.PIPESIZE);
                   offset+=count;
               }
               else if((bout.length-offset)<UtilsExternal.PIPESIZE){
                   u.writePout(bout,offset,bout.length-offset);
                   offset=bout.length;
               }
            }
        }
    }
}

/**********************************/
 * LabComm.Serial.UtilsSerial    *
/*******************************/
package LabComm.Serial;

/**
 * The <code>UtilsSerial</code> class is a util class that contains methods
 * for converting serialized objects to arrays of bytes
 * and vice versa.
 * It works using the interface Serializable and let every serializable object to
convert its content
 * into array of bytes. While reading, the object is reconstructed from this content.
 * @author
 */

public final class UtilsSerial {
    /**
     * Private buffer to store data.
     */
    private byte[] b=new byte[BUFFERSIZE];
    /**
     *Boolean used to syncronize methodes write and read of this class.
     */
    private boolean availableRead;
    /**
     * Piped output stream used for conversion.
     */
    private PipedOutputStream pout;
   /**
    * Piped input stream used for conversion.
    */
    private PipedInputStream pin;
    /**
     * Object output stream used for conversion of objects to byte arrays.
     */
    private ObjectOutputStream out;
    /**
     * Object input stream used for conversion of byte arrays to objects.
     */
    private ObjectInputStream in;
    /**
     * The Size of the piped
     */
    static final int  PIPESIZE=1024;
    /**
     *The size of the auxiliar buffer, it has been modified if it is necessary.
     */
    public static final int  BUFFERSIZE=128000;
    /**
     * Constructs an utility object and initializes all streams.
     * @exception IOException if an I/O error occurs.
     */
    public UtilsSerial() throws IOException {
       initialize();
    }

    /**
     * Initializes the streams.
     * @exception IOException if an I/O error occurs.
     */
```

```
private void initialize() throws IOException {
   pout = new PipedOutputStream();
   pin = new PipedInputStream(pout);
   out = new ObjectOutputStream(pout);
   out.flush();
   in = new ObjectInputStream(pin);
}

/**
 * Writes the buffer in the auxiliar and notifies it.
 *@parameter buff
 */
public synchronized void write( byte[] buff){
   b=buff;
   availableRead=true;
   notifyAll();
}

/**
 * Writes length number of bytes of the buffer starting from offset in
 * PipedOutputStream.
 * @param buff It is the buffer from whom we take the information.
 * @param offset
 * @param length
 * @exception IOException if an I/O error occurs.
 */
public synchronized int  writePout( byte[] buff,int offset,int length){
   try{
       pout.write(buff,offset,length);
   }catch(IOException e){System.out.println("Inside utils writePout"+e);}
   return length;
}

/**
 * Returns private buffer b.
 * @return a byte[].
 */
public synchronized byte[] read(){
   try {
       while(!availableRead) wait();
   }catch(InterruptedException e){System.out.println(e);}
   return b;
}

/**
 * Reads from PipedInputStream and stores the bytes in the buffer b.
 * @return the number of bytes read.
 * @exception IOException if an I/O error occurs.
 */
public int readPin(byte[] b,int offset,int length){
   int bytes=-1;
   try {
       bytes=pin.read(b,offset,length);

   }catch(IOException e){System.out.println("Inside utils readPin"+e);}
   return bytes;
}

/**
 * Converts an object to an array of bytes using piped streams.
 * @param s the serialized object to convert.
 * @return the array of bytes that represents the serialized object.
 * @exception IOException if an I/O error occurs.
 */
public  void convert(Serializable s) throws {
   out.reset();
   out.writeObject(s);
   out.flush();
}

/**
 * Returns a buffer of bytes where the object already serialized is stored.
 * This method uses ReadThreadS to avoid the size limitation of ObjectInput and
 * ObjectOutput. The object must implement serializable.
 * @param s
 * @return byte[]
 * @throws IOException
```

```
     */
    public  byte[] convertE(Serializable s) throws IOException {
       byte[] b;
       ReadThreadS rt;

       rt=new ReadThreadS(this);
       rt.start();
       convert(s);
       // When there is not more data to be read, the readT thread is interrupted
       boolean data=true;
       while(data){
           if(this.available()!=0)
              data=true;
           else{
              data=false;
              rt.interrupt();
           }
       }
       // Writes in the byte[] b the array which has been created in readThread
       b=read();
       return b;
    }
    /**
     * Calculates the size of an serialized object, in bytes.
     * @param o the serialized object.
     * @return the size of the serialized object.
     * @exception IOException if an I/O error occurs.
     */
    public int getObjectSize(Serializable o) throws IOException {
       return convertE(o).length;
    }

    /**
     * Returns the number of bytes that can be read from PipedInputStream.
     * @return the number of bytes that can be read from PipedInputStream.
     * @exception IOException if an I/O error occurs.
     */
    public int available(){
       int bytes=-1;
       try{
           bytes=pin.available();
       }catch(IOException e){System.out.println(e);}
       return bytes;
    }
}
```

```
/**********************************/
 * LabComm.Serial.ReadThreadS    *
/*********************************/

package LabComm.Serial;

/**
 *This thread reads from the PipedInputStream and save all the information in a byte
array,
 *All data is stored in a private buffer utils.java,
 *TransmiterThread will read this buffer and it will send its content.
*@author
*/

public class ReadThreadS extends RTThread{
    private byte[] b=new byte [UtilsSerial.BUFFERSIZE];
    private UtilsSerial u;
    private byte[] pack;
    /**
     * Constructor
     * @param ut
     */
    public ReadThreadS(UtilsSerial ut){
       u = ut;
    }
    public void run(){
       int offset=0;
       int bytesread;
       //It is interrupted by the TransmiterThread, when there are not more data in the
       pipeInputStream
       //and TransmiterThread has finished to extarnalize dates( writes dates in the
       pipeOutputStream).
       while(!isInterrupted()){
           //Reads only when there are data in the PipedInputStream
           if(u.available()!=0){
              bytesread=u.readPin(b,offset,UtilsSerial.PIPESIZE);
              offset+=bytesread;
           }
       }
       //pack array is used to send only the significatives bytes,
       //since b has a long size many times it is not full
       //and there are many zero values in the last positions of the array.
       pack=new byte[offset];
       for(int i=0;i<offset;i++){
              pack[i]=b[i];
       }
       //Writes pack array so that TransmiterThread can read it and transmit it after.
       u.write(pack);
    }
}
/**********************************/
 * LabComm.Serial.WriteThreadS    *
/*******************************/
package LabComm.Serial;

/**
 * This thread writes the received byte array in a PipedOutputStream.
 * The information is written in blocks of 1024bytes, except for the last block that has
 * the bytes left.
 * @author
 */
public class WriteThreadS extends RTThread{
    byte[] bout=new byte[UtilsSerial.BUFFERSIZE];
    UtilsSerial u;
    /**
     *Constructor
     *@param b byte buffer
     *@param ut UtilsExternal
     */
    public WriteThreadS(byte[] b,UtilsSerial ut){
       bout=b;
       u=ut;
    }
    public void run(){
       int offset=0;
       int count=0;
       //it is interrupted by the receiver thread when this finishes externalize method.
       while(!isInterrupted()){
           if(offset<bout.length){
              if((bout.length-offset)>=UtilsSerial.PIPESIZE){
                  count=u.writePout(bout,offset,UtilsSerial.PIPESIZE);
                  offset+=count;
              }
              else if((bout.length-offset)<UtilsSerial.PIPESIZE){
                  u.writePout(bout,offset,bout.length-offset);
                  offset=bout.length;
              }
           }
       }
    }
}
```