A DISTRIBUTED FRAMEWORK TO FACILITATE HUMAN-ROBOT REMOTE INTERACTION

by

AHMAD TAWFIQ AL-HAMMOURI

Submitted in partial fulfillment of the requirements

for the degree of Master of Science

Thesis Advisor: Dr. Vincenzo Liberatore

Department of Electrical Engineering and Computer Science

CASE WESTERN RESERVE UNIVERSITY

January, 2004

Signature Sheet

Dedication

To my loving parents, brothers, and sisters.

Table of Contents

List of Figures

List of Examples

Acknowledgements

I would like to thank my thesis advisor, Professor Vincenzo Liberatore, for his endless encouragements, help, and advice during the work on this thesis. It is a privilege to work with you.

Special thanks to Professor Wyatt Newman and Adam Covitch for the invaluable discussions we had during our group meetings.

At last but not least, I would like to thank my colleagues in the Network Research Lab and my roommates Abdullah Jordan, Huthaifa Al-Omari, and Osama Al-Khaleel.

Thanks, everyone.

Ahmad Al-Hammouri, November 2003

A Distributed Framework to Facilitate Human-Robot Remote Interaction

Abstract

by

AHMAD TAWFIQ AL-HAMMOURI

In this thesis, I present a distributed multi-agent software system that facilitates remote interaction between human beings and intelligent systems, e.g., robots. This system is built using off-the-shelf open-source and free-licensed software technologies. The two major technologies used are Aglets, which are Java-based mobile agents, and Jini, which is a middleware for building adaptable and dynamic distributed systems.

The resulting system has the following features. First, it allows the user to expand the functionality of a robot on the fly. Second, it allows the user to control multiple robots from one control panel. Third, in face of partial failures, the system still functions in a consistent and predictable manner.

I discuss the system's different components and present experiences and directions for future research.

Chapter One

Introduction and Background

1.1 Introduction

The Internet provides a suitable infrastructure for building distributed applications. This is because computer networks, which make up the Internet, are becoming faster and relatively more reliable [14]. One of these applications is Internet Tele-robotic. The first experiment to interact and control a robot remotely was the Mercury project [11] at the University of Southern California. Following this experiment, many robotic research groups exposed their own robots to the Internet audience. To name a few are the Tele-garden project [12], Xavier [24], and the efforts done at the University of Western Australia [25]. The audience found the idea a very fascinating one because no matter how far they were from the robot's actual location, they could still interact with and manipulate the robot.

The growing interest in this field of research is seen to be useful in areas where the human access to a place is hazardous, or because of physical separation. For example, robots can be used in nuclear plants or sent to space.

In this thesis, we present a distributed multi-agent software system that facilitates remote interaction between human beings and intelligent systems, e.g., robots. This system is built using off-the-shelf open-source and free-licensed software technologies. The objective is to build a system that is:

- Environment independent. The system should not be restricted to run on a special hardware or operating system. In addition, if it needs to be moved from one

environment to another, no changes or minor changes should only be required; and

- Generic. The system should not be based on a particular robot and system components should be abstracted to fit a wide range of robots.

The road map of this thesis is structured as follows: the remaining of this chapter surveys a number of pervious and related works and introduces a background of our approach. Chapter 2 discusses in details the anatomy and the building blocks of the system. Chapter 3 gives different techniques implemented to increase the system robustness and reliability. Chapter 4 gives an example of how the system can be tailored to interact with a specific robot through a number of screen shots. Chapter 5 concludes the thesis and presents directions for future works.

1.2 Related Work

This section presents an overview of related research projects. It is not intended to evaluate the robot's hardware for each, but to analyze the system used to control each robot.

1.2.1 Previous Work

The Mercury system [1] consists of a robot, a camera, a UNIX station, and a PC. The camera is used to capture and broadcast images from the robot's workspace. The two machines, the UNIX station and the PC, are connected via Ethernet. The UNIX station, which runs a web server, is responsible for accepting and responding to the incoming HTTP requests, verifying users' Ids and passwords, decoding mouse events into XY coordinates and then sending them to the PC, and accepting the feedback images from the PC and sending them to the users. The PC, which is connected to the robot via a serial port, is responsible for decoding the XY coordinates into robotic commands, sending these commands to the robot, and receiving the workspace images from the camera, compressing them, and then delivering them to the UNIX machine.

When users direct their web browsers to the project's URL, they receive a web page that contains a status image and three buttons. Two buttons are used to move the camera to give a different view of the workspace and the third is used to blow a burst of air into the sand underneath the robot's arm to look for a hidden object. Users can also click the mouse on the image to instruct the robot to move to a specific location.

The System presented in [17] uses CORBA as a communication protocol between the client and the server. CORBA is used to enable a Java code to communicate with a

legacy C++ code. Accessing the project's web page with a Java-enabled web browser, clients download a Java applet interface that allows them to interact with the robot. The interface has the following components: an object recognition module, a live image of the robot's workspace, and a graphical representation of the robot's current state. This representation is to be used as a feedback if only low bandwidth is available.

The user interface of the system described in [13] is based on a Java applet that communicates with a Java servlet running on the web server. The interface has two components: a JPEG image, which displays the live robot actions, and a VRML graphical model. The VRML model is used to move the robotic arm and to serve as a feedback representation of the actual robot's state.

Xavier, an autonomous mobile robot, is described in [24]. It deploys complex algorithms, such as obstacle avoidance, navigation, and path planning. The web-based interface allows users to select one of many predefined locations within a building. After determining the efficient route to the commanded location, Xavier follows this path to the specified location. The interface also has an image showing the robot's status.

A reusable framework for web-based teleportation is presented in [8]. Built to be as generic as possible, the system targets a wide range of devices to make them available to the Internet audience. This system can be used to allow more than one user to manipulate more than one robot at the same time. The architecture consists of the following building blocks:

Device Server: The device server, which represents the wrapper to the device (robot), accepts robotic commands from the web server and forwards them to the robot. It was implemented using Expect scripts, an extension of Tcl scripting language.

Web Server: The web server is based on Python Medusa web server. In addition to its functionality of publishing the robot on the Internet, it coordinates the communication between users and device servers.

Remote Viewers or Pilots: These are the clients, who interact with the robots.

The system is made adaptable by using the idea of configuration files and the JavaBeans Technology, which is used to build the user interface. The device (robot) administrator is responsible for tailoring the adjustable components to fit a specific device.

A distributed framework for Internet robotics is discussed in [4]. It uses XML as the communication protocol and Message-Oriented Middleware (MOM) [23] as the underlying architecture. The system entities or peers – the web server, the robot, the camera server, and the clients – are connected to a central message-router, which MOM provides. Changes in the system's state are sent to peers as events. These changes include the cases when peers join and leave the system. The user interface has four elements: a robot panel, a camera panel, a user chat panel, and a console panel. The robot panel lists the robot commands that users have submitted and allows new commands to be typed and sent to the robot. The camera panel shows live images from the robot's workspace. Users can select different views and can change the picture's zoom and attributes. The chat panel allows the users, who have already logged in the system, to chat and collaborate to accomplish some robotic tasks. The console is used to display status and error messages.

Several distributed robotic systems, such as in [1, 9, and 18], rely on Real-Time CORBA [22], an extension of CORBA [27], to meet the time-delay constraints. The overall goal of Real-Time CORBA is to ensure end-to-end predictability in real-time

systems [30]. Using CORBA as a communication standard, the resultant system is platform independent and different parts of the system can be programmed using different programming languages. When these systems need to be extended beyond firewalls, HTTP tunneling is used.

1.2.2 Summary

The systems mentioned in the previous section share one aspect – they all follow the client-server computing paradigm. Figure 1.1 shows the general coordination of these systems.

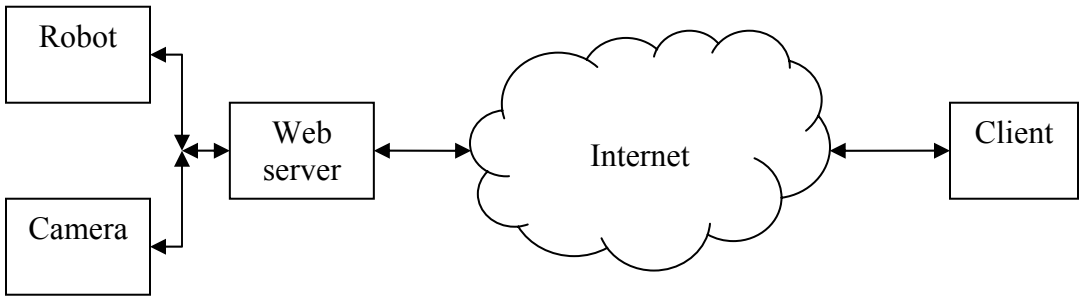

Figure 1.1: A general coordination of Internet robotic systems.

The sever, which is usually a commercial Web server, publishes the robot on the Internet, accepts requests and robotic commands from clients, forwards the robotic commands to the robot, receives the reply from the robot and the status image from the camera, and delivers the feedback to the clients. The clients, on the other side, download an interface that enables them to send commands to the robot and receive the robot's current state images.

What differentiates one system from another is the programming language or the technology that is being used. One system may use the Java programming language and its technologies, such as Applets, Servlets, and RMI; another may use CGI processes at the server side and carry out the whole communications using CORBA, whereas a third may replace the robot's feedback image with a 3D representation that renders at the client side.

Such systems demonstrate the feasibility to teleoperate robots over the Internet [19]. They still, however, lack one or more of the following important requirements – which prevent them from being applied to a wide range of applications:

- On-the-fly programmability. It is desirable to give the robot's user the ability to construct new tasks based on the currently available ones. For example, the functionality of a mobile robot that supports just a single command, move to XY location, can be extended to another command like move in a square shape, which consists of four "move to XY location" commands invoked in sequence with proper parameters. The system should allow the user to create, inject, and invoke the new functionality on the fly, i.e., while the system is running.
- Multi-robot control. If more than one robot is available, the user may need to control them simultaneously and to distribute the work among them. Providing the user with this ability, the system needs to be generic to encompass different kinds of robots.
- Partial failures tolerance. Network connections may fail; machines, devices, and software programs may crash during robot operation times. The importance of how to deal and withstand these problems relates to the importance of the task the

robot is doing. Although this is a secondary issue, or perhaps it is not an issue at all, for a robot being used to play with colored marbles, it is, indeed, a critical issue for a robot being used to perform surgery.

- Latency alleviation. Communications over the Internet experience long and time-varying delays. The amount of this delay depends on network congestion, link bandwidth, and source-to-destination distance [3]. Designers of Internet-based telerobotic systems must account for latency to ensure the effectiveness of robot control [15].

The next chapter presents an implementation of a distributed framework to achieve these important requirements.

1.3 Background of a new architecture

The work presented here is based on the architecture introduced in [21]. This architecture relies on flexible software entities called Virtual Robots (VRs). The VRs have the following characteristics:

- They are mobile software agents; therefore, they have the ability to migrate from one machine to another and to communicate with one another.
- They are logically deployed between the supervisor (user) and the robot under control.
- The supervisor creates them on demand to wrap the functionality of a robot or of another VR into a higher-level coarse-grained functionality.
- Their overall structure forms a tree, where the user controls its root and the physical robots form its leaves.

The first prototype of this distributed architecture was implemented using the C# programming language in the .NET environment. Anyone who wants to deploy this system and to extend its functionality has the flexibility to use any of the languages supported by the .NET environment – for example C, C++, C#, and Visual Basic. Because the system uses SOAP [2] as the communication medium, it can work behind firewalls and can span LANs and WANs.

This prototype achieved a subset of the preliminary proposed goals. Each of the remaining parts represents a stand-alone project. For example, providing code mobility to C# programs is by itself a separate project. We, therefore, have decided to re-implement another version of the same system using the Java programming language. We have made

such a decision because Java has the following advantages – none of them currently existing in the .NET environment:

- Environments for developing Java programs are available on the World Wide Web (WWW) for anybody and at no cost.
- The Java code written on one platform can be transferred to a different platform without changing anything. Thus, the resultant system can span many heterogeneous environments.
- Many Java technologies and implementations for distributed systems are available free on WWW. These technologies, which are built by professional groups, will ease the burden of developing various system parts. Therefore, the effort can be directed to develop and enhance other sides of the system.

The Java technologies used in the system are Aglets, Jini, and Java Spaces. An overview about each is presented next.

1.3.1 Aglets

Aglets are Java objects that support the concept of mobile agents. The Aglet API [16 and 26] is a development kit that allows a developer to create a new aglet as an instance of a Java class, clone a twin aglet from an existing one, dispatch an aglet to another machine, retract an aglet from another machine, and dispose an aglet when there is no longer need for it.

We use Aglets to build the Virtual Robots (VRs); therefore, VRs can move from one machine in a network to another. This ability to travel offers a solution for network

latency; hence, helps real-time systems to fulfill time constraints during communication [16]. Figure 1.2 demonstrates this benefit.

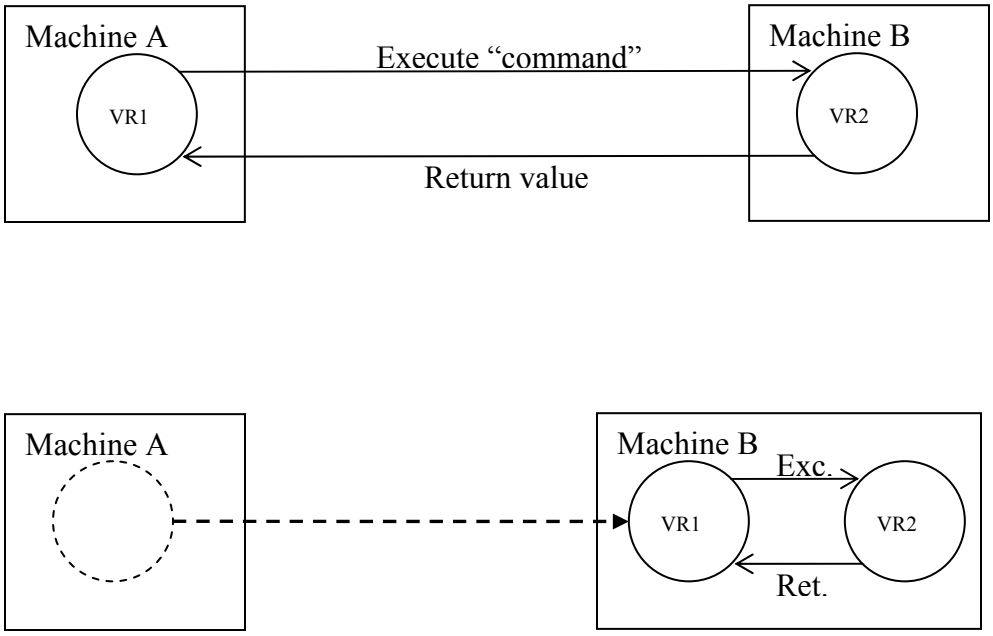

Figure 1.2: The benefit of VR mobility. (Up) VR1 is in machine A; VR2 is in machine B. (Down) Both VR1 and VR2 are in machine B.

Instead of having commands and return values travel through the network medium connecting machines A and B, as shown in the upper part of figure 1.2, VR1 can migrate to machine B, lower part of figure 1.2. Thus, VR1 and VR2 will execute as two threads in the same machine and will communicate using inter-thread communication, which is far faster than network communication.

Other advantages of VRs’ mobility are fault tolerance and load balancing. If a VR realizes that its current machine is shutting down or the available computation resources are stepping down, it can move to another machine with suitable resources.

1.3.2 Jini and Java Spaces

Jini, a middleware built on Java and RMI, provides a network infrastructure and programming model to build adaptable and dynamic distributed systems [1, 6, 20, and 28]. It eases the implementation of distributed applications by introducing the concepts of lookup and discovery, leasing, distributed events, and objects movements over a network. It also enables the deployment of Java Spaces [7] technology, which is used to render the system's persistence.

The next two chapters explore the anatomy of our system and explain how all of these concepts are implemented in the system.

Chapter Two

Architecture Exploration

In this chapter, a thorough discussion of our infrastructure is presented.

2.1 System Requirements

To apply the system to a specific robot effectively, the following two requirements should be met:

- The exact type of the robot itself is irrelevant. However, it should have the capability to endure the communication delays introduced by the network. This requirement is essential to ensure the robot arrives to a safe and predictable state in face of long delays, lack of communication, jitter, or partial failures during operation times.
- The machine to which the robot is directly attached should co-exist with other machines connected by a network. By a machine, we mean any device that is able to store and run Java programs and to communicate with others over TCP/IP. This may be a PC, a laptop, a UNIX machine, or as simple as a PDA. These machines should be accessible by the organization operating the robot since they must run daemons and accept mobile code.

Communications among various system entities use TCP/IP. TCP/IP has proven its strength in Internet. We rely on TCP/IP reliable data delivery service – i.e., the transmitted data is guaranteed to arrive at the destination intact and in order – to ease the

implementation of system entities. It is acceptable that TCP/IP does not make any guarantees and bounds on minimum transfer rates and maximum network delays, but this problem is eliminated by the robot controller. The next section elaborates on this problem. The system is implemented within the application layer of the Internet protocol stack and it is relying on TCP/IP protocol but if it happens that a new, state-of-art and more powerful transfer protocol is invented, we can adapt to it without any difficulty provided that this protocol conforms to the transport layer in the Internet protocol stack.

2.2 The time delay issue

We believe that we have to deal with the communication time-delay problem because, sometimes, there is no means to avoid it no matter what the software in the application layer, the transport protocol, or the physical communication medium we use. Take an example the robot that is on Mars and is controlled from Earth. In the best case, the data will travel at the speed of light and in a single direct link. Knowing that the shortest distance between Earth and Mars is 33 million miles (53.1 million Km) and the longest distance is 249 million miles (401 million Km) [29], one can calculate the minimum time-delay (t_{min}) and the maximum time-delay (t_{max}) as*:

$t_{min} = 53.1*10^6 / 3*10^5 = 177$ seconds or 2.95 minutes, and

$t_{max} = 401*10^6 / 3*10^5 = 1336.67$ seconds or 22.3 minutes.

The average of both values is 12.63 minutes. Therefore, we need 25.26 (12.63*2) minutes, on average, to send a data packet to Mars and receive a confirmation that it has, indeed, arrived. So what have we done to deal with the communication time-delay problem?

First, as mentioned previously, we rely on the robot controller to encapsulate the real-time constraints. Natural Admittance Control (NAC) realizes this requirement and it guarantees robot stability in absence of QoS provisioning. More details about NAC are in [5, 10, and 21]. Eventually, the communication time delay will affect just the time needed to complete a specific task. Second, we assume that the robot has some degree of autonomy to be driven in supervisory control mode. In our architecture, the robot controller does not have to support this autonomy by itself, but by means of the enslaving

* The speed of light is $3*10^8$ m/s or $3*10^5$ Km/s

(encapsulation) technique, which is discussed later, we will let some VRs wrap the robot functionality into a higher granularity. The VRs will accept a high-level coarse-grained command, break it into smaller ones, and then deliver them to the robot in a granularity it understands. These VRs should live near the robot, though. By near, we mean that VRs are on the same machine or on machines close to one another so the communication delay among them is very small, e.g., they are on machines in the same LAN. Finally, we equipped VRs with the ability of migration from one machine to another. A VR can move to a machine that is near another VR or the robot .This was explained in figure 1.2 in chapter 1.

2.3 Software Entities:

The system is based on the following software entities:

2.3.1 Robot Proxy (RP)

To embed any physical robot in our system, an interface is required to hide the robot's proprietary hardware and its controller's software. The RP represents a surrogate or a shield between diverse implementations of different robots and our system, which is based on Java. From one side, the RP is compatible with other system's entities and it delivers all the services as if it is the robot software itself. From the other side, it communicates with the robot's proprietary software to handle to it the desired commands. The RP communicates with other system entities by Aglet messaging (see below) and with the robot's software by sockets. The RP resides on a machine close to the robot and keeps an active socket connection with the robot's software.

The program at the robot, which is written by the robot's manufacturer specific language and communicates with outside world, is referred to by "RPCS" (Remote Procedure Call Stub), which usually accepts ASCII-based messages and maps them to the robot's hardware. The following diagram depicts these elements:

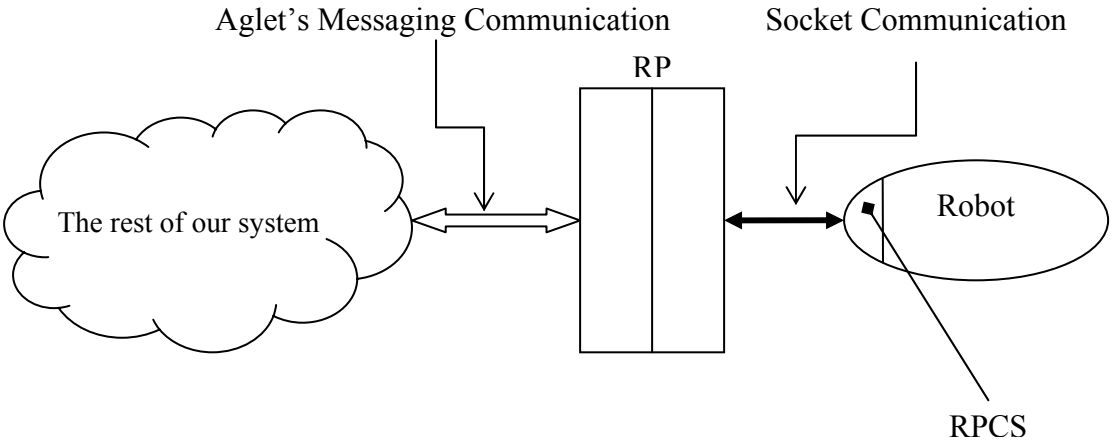

Figure 2.1: The Robot Proxy's (RP) role in the system.

Aglet Messaging:

Aglet messaging is a message-based communication technique among aglets. For an aglet, A, to send a message to another aglet, B, aglet A has to have aglet B's proxy. The proxy, which serves as a message gateway for the aglet, is an object that has the complete information about the aglet location, identity, and state. Sending a message to an aglet is simply calling the "`sendMessage`" method on the aglet's proxy object.•

There are two flavors of aglet messaging: *synchronous and asynchronous* messaging. When the sender uses synchronous messaging, its thread of execution blocs

• The aglet proxy can also be used to clone, dispatch, and dispose the corresponding aglet.

until it receives a reply. Sometimes, the sender specifies not to receive a reply. In this case, the sender sends the message and continues execution – this is called one-way-type messaging.

In asynchronous messaging, the sender expects to receive a reply, but it does not block its execution. Another name for this type of messaging is future-type messaging where the sender sends a message and continues execution and when the reply arrives, the sender starts working on it. There are methods to check whether the reply has arrived, to wait a certain period for a reply to arrive, and to retrieve the reply from the queue once it arrives.

Each aglet that expects to receive messages must extend the "`handleMessage`" method. This method can be customized to accept and take action based on particular kinds of messages. Different kinds of messages can be given different priority levels, which will ensure that messages with higher priority levels are handled before those with lower levels.

2.3.2 Virtual Robots (VRs):

The Virtual Robots are mobile software agents that have the ability to move from one place to another. They communicate among themselves and other entities via Aglet messaging. Any VR has the behavior to enslave or encapsulate the functionality of another VR or RP. By this mechanism, the functionality of the robot can be extended to a new one. This resembles the inheritance in software technology. The following example clarifies the relationship between the enslaving and the enslaved VRs. Suppose that VR3 enslaves VR2 while VR2 enslaves VR1 as shown in the diagram below:

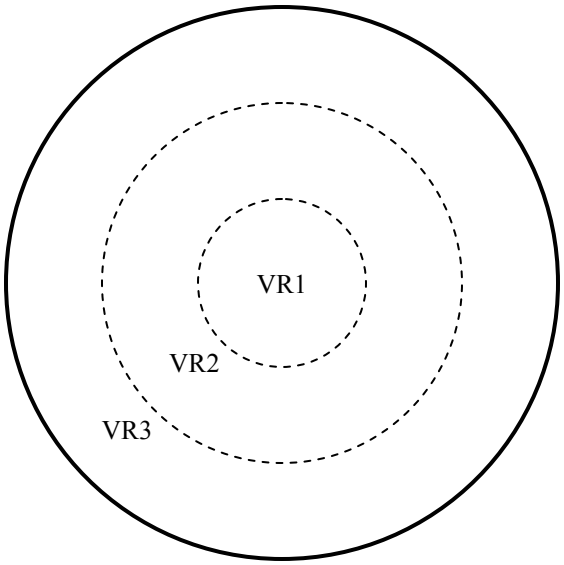

Figure 2.2: Virtual Robot's Encapsulation.

A user, who is dealing with VR3, sees just the functionality supported by VR3 and the functionalities inherited from VR2 and VR1, which VR3 explicitly declares. The user is unaware of that VR3 enslaves VR2 and VR2 enslaves VR1, and he thinks that he manipulates the physical robot through VR3. Hence, the name Virtual Robot comes.

A user can create, enslave, and dispose (destroy) a VR using the Virtual Supervisor.

2.3.3 Virtual Supervisor (VS):

The Virtual Supervisor is an agent that works on behalf of the person (supervisor) who wants to manipulate a robot. The job of the VS, once it is started, is to find all lookup services existing in the system and to ask them for any registered entities of type Robot. By an entity of type robot, we do not mean the physical robot but rather we mean any software component that has the ability to accept robotic commands, which includes all VRs and RPs. After getting the answer of the currently available entities, the VS asks the lookup services to inform it back of new VR or RP arrivals. The lookup services use event notifications to keep the VS updated with changes happen in the system.

The VS is simply a GUI that facilitates the interaction with system's other components. It gives the supervisor the ability to create new VRs, enslave them, contact them to execute robotic commands, and delete them.

2.3.4 Lookup Services (LUs):

The LU is a directory that contains information about VRs and RPs available in the system. Newly created VRs and RPs contact the lookup service to advertise themselves to others.

Multiple LUs can be used and distributed around the system to provide redundancy and to ensure the robustness, such that, if one LU fails, others can backup its functionality. Not all LUs have to be running before other system entities; it is sufficient that one, at least, to be running beforehand. When others are then started, they will get the whole information from the LUs that are already running.

2.3.5 Visual Feedback

There are two forms of feedback when interacting with a robot. First, the low-level feedback, which represents the return values from executing robotic commands, such as, position coordinates, force values, status messages. These values take form of numbers, strings, and other data types. Second, the high-level feedback, which shows the robot and its surroundings. This feedback takes form of moving pictures or live video.

The deployment of the visual feedback in our system mimics exactly the deployment of the RP. As the robot administrator starts up the RP to publish the availability and the functionality of the physical Robot, he or she also starts another program, which has a convenient way to showing the workspace of the robot and its actions in terms of a streamed video, discrete moving pictures, or 3D graphic simulations to users.

When the VS contacts the LUs to get a list of available objects of type robot, it will get also a list of available objects of type “Workspace View”. The “Workspace View” refers to any entity registered with LUs that has the property to give a visual scene of a robot place. Figure 2.3 shows the job of VS at startup.

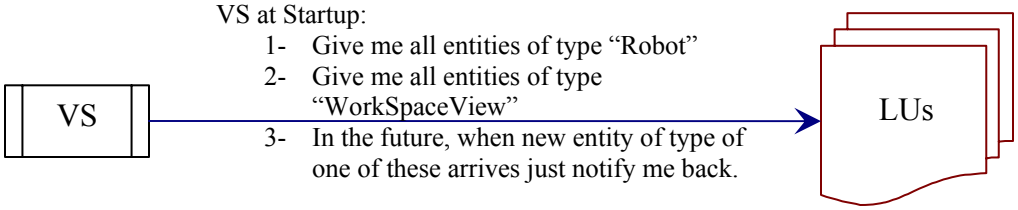

Figure 2.3: VS at startup.

The user can select a view, press a button in the VS GUI, and then a live scene will pop up on the screen. The internal implementations of how this works are hidden from the user and are specific to the device (the camera) that is being used. The robot's administrator is the one who takes care of this.

Unlike the RP, the workspace view does not have the property to be enslaved and users cannot extend its functionality.

2.4 Putting It All together (The complete system)

After presenting the building blocks of the system let us put them all together and see how the final coordination looks like. In general, we will have multiple physical robots being controlled by single VS. We will have also many VRs forming a tree to support this control. The diagram on the next page (Figure 2.4) shows a broad and general example of such layout.

The RPCS and the RP are two layers used as an interface between the physical robot and the remaining system entities. Each physical robot has its own RPCS and each RPCS has its own RP, so if it happens that a new Java-enabled robot needs to be introduced in the system, neither RP nor RPCS is needed. Figure 2.4 can be redrawn using the substitution shown in figure 2.5.

The tree shown in figure 2.4 builds up over time and it does not reach this point in a single shot. It may also grow up into a more complex shape with additional time. Let us have a possible scenario during the course of building up this tree.

At the very beginning, when a new robot is brought into the system, its corresponding RP registers with LUs.

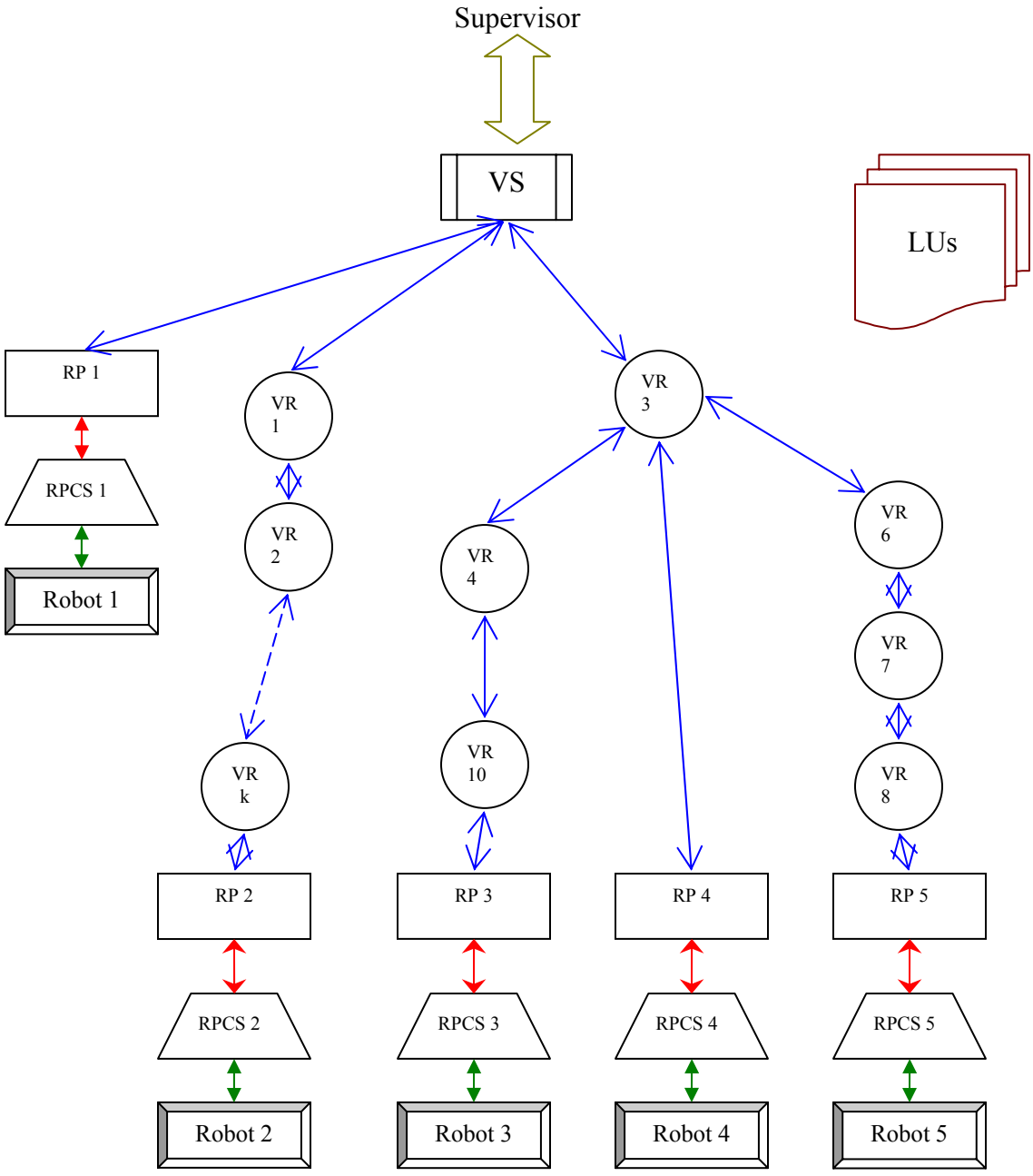

Figure 2.4: A very broad and general class of scenarios.

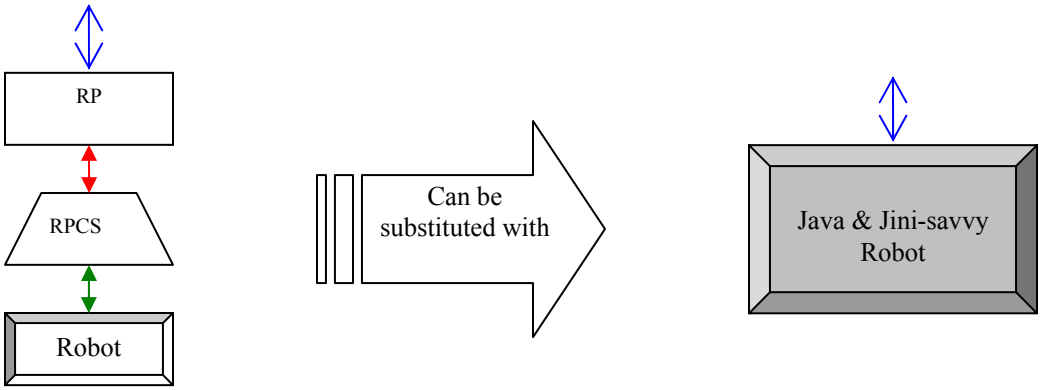

Figure 2.5: Considering RP, RPCS and the physical robot as one entity.

The VS gets aware of this and adds it to the available robot list. Then the supervisor can interact with this robot through its proxy (the RP), as the case with "Robot 1" in figure 2.4. After the supervisor becomes familiar with the robot's functionality, he decides to expand those low-level commands into a useful high-level one comprises one or more of the already defined ones. As the time runs, new ideas and new tasks can be added until we have a chain-like set of VRs, as the case with "Robot 2" and VR1. Sometimes, it is desirable to consolidate the functionality of two or more separate robots and control them by a single super node, as shown in figure 2.4, VR3 has a mix of functionalities inherited from VR4, VR6, and RP4.

In the layout seen in figure 2.4, the VS can interact with RP1, VR1, and VR3. It can issue tasks to VR1 and while a task is being carried out the supervisor can issue tasks to VR3 or RP1 and there is no need to set back waiting VR1 to be done with its commanded task.

As was stressed out, the VS (or the supervisor) has no access to the inter-tree VRs or RPs, e.g., the VS has no knowledge of what functionality VR2 supports, and cannot delete, say, VR6 before deleting VR3 first. However, it can view the tree structure of such layout and see how entities are connected.

2.5 Software Classes

Here, we present the Java classes that contribute to building up the system. The definition and the purpose of each are given.

2.5.1 Common Classes

These classes are found in VS, VR, RP, and WorkSpaceView. They define the communication protocol between parties. They are written as Java Interfaces, so they tell what needs to be done but do not dictate how it should be done (the internal implementation). These classes are:

2.5.1.1 Robot Interface

This interface defines how the VS interacts with RPs and VRs. RPs and VRs implement (extend) this interface and override its methods. One RP may implement this interface in a different way from how another RP or VR may do and still the VS does not have to know about the internal implementation of each VR or RP. The VS just invokes the defined methods and gets the defined return values. This makes the system generic and adaptable to many RPs and VRs developed by different people at different times. The Robot interface declares methods to get a handle to communicate with a specific VR or

RP, to know the type of a VR or RP, and to ask a VR or RP for the TaskGUI object (explained in section 2.5.2.4).

2.5.1.2 WorkSpaceView Interface

This interface is similar to the previous one but it has to do with the visual feedback, which is built using the same idea the RP is built. The VS just calls a method like "`getView()`", then a frame will show up having the robot's scene. It is totally up to the implementer how to extend the interface and how to implement the method. As a result, different views may be implemented in different ways.

2.5.2 Robot Proxy classes

These include:

2.5.2.1 RobotProxyProxy class

The RobotProxyProxy class implements (extends) the Robot interface. Before interacting with a specific RP, the VS downloads the corresponding RobotProxyProxy object from the LUs. This object is knowledgeable about and communicative with the backend entity – the RobotProxy.

2.5.2.2 RobotProxy class

This is the backend entity, which is responsible for the following:

- At startup, it establishes a socket connection with the RPCS for future communication.
- It registers with available LUs by uploading an instance of RobotProxyProxy class.
- It waits for incoming commands. These commands could be system control commands, such as, registration and deregistration with LUs, and requesting the TaskGUI object (see section 2.5.2.4); or they could be robotic commands, such as, go to a place, catch an object, rotate, and others supported by the robot. Robotic commands are then sent through the socket to the RPCS.

The RP has the ability to recover if its parent VR crashes unexpectedly. We will come to this issue later in chapter 3.

2.5.2.3 Messenger

The Messenger is an Aglet that carries the TaskGUI from the RP and migrates with it to the VS. At arrival, it displays the GUI at the supervisor's computer screen, so by then he or she can issue robotic commands.

2.5.2.4 TaskGUI

The Messenger Aglet when arriving at the VS machine creates an instance of this class. The TaskGUI class is written by the robot's administrator to reflect the robot functionality in terms of convenient GUI components. As the user interacts with the GUI, messages having robotic commands are sent to the RP accordingly.

2.5.3 Virtual Robot classes

Most of these classes are similar to those of Robot Proxy's classes. Only differences are described here.

2.5.3.1 VirtualRobotProxy, Messenger, and TaskGUI classes

These three classes have the same properties of RobotProxyProxy, messenger and TaskGUI found in Robot Proxy classes. The VirtualRobotProxy communicates backend with an object of a class that extends the VirtualRobot class instead of a RobotProxy object. The TaskGUI reflects the new functionality of the VR, so the user, who builds the VR, writes the appropriate TaskGUI class.

2.5.3.2 VirtualRobot class and its subclasses

The VirtualRobot's class is the base class of all VRs entities. All VRs extend (inherit from) this class. This class has all the functionalities to enable each VR to be created, to enslave other VRs or RPs, to be enslaved by another VR, to migrate to another machine in the network, and to be disposed. Later in the thesis, we cover how a subclass can extend this class to form a useful VR.

The VR is equipped to perform the following duties:

- At creation by the VS, VR registers itself with all available LUs by uploading to them an object of VirtualRobotProxy class.
- Upon request from the VS, VR delivers its GUI to the VS by creating an instance of class Messenger, loading it with an instance of TaskGUI class, and dispatching it to the machine where VS resides.
- VR can enslave other VRs or RPs and enrich their functionalities.
- When being enslaved by another VR, VR deregisters itself from the LUs. If its parent is deleted, it reregisters again.
- When moving, VR updates other VRs about its new location.
- When disposed, VR notifies other system's entities, such as, other VRs, RPs, or LUs about its willingness to disappear.

2.5.4 Virtual Supervisor classes

Besides the common classes mentioned earlier, the Virtual Supervisor has many inner classes to support its functionality.

The VirtualSupervisor's main duty is to display the system components in a graphical and interactive way to the supervisor. The GUI has five tabbed panels. The first panel (figure 2.6) shows a list of available VRs and RPs in the system and gives the user the ability to request each VR or RP TaskGUI and to enslave VRs or RPs. The second tabbed panel (figure 2.7) enables the user to navigate the machine's file system to select a class file to create a new VR. The third panel (figure 2.8) enables the user to delete a specific VR. The fourth one (figure 2.9) is similar to the first, except that the list is a list of Workspace views, so the user can press a button to see a live scene for a robot. The fifth panel (figure 2.10) enables the user to see the tree structure of VRs and RPs.

Chapter 4 shows how to use these GUI panels to interact with RPs VRs, and Workspace Views.

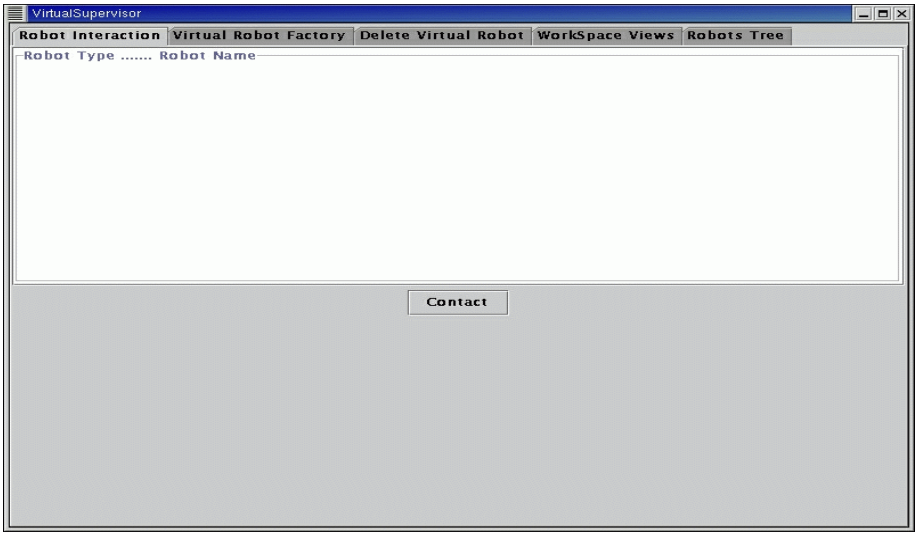

Figure 2.6: VS GUI screenshot: panel 1.

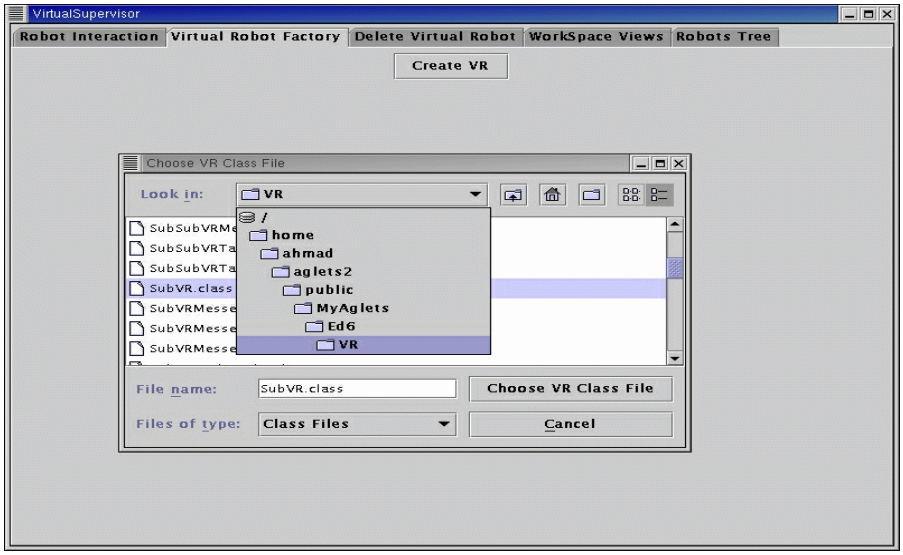

Figure 2.7: VS GUI screenshot: panel 2.

Figure 2.8: VS GUI screenshot: panel 3.

Figure 2.9: VS GUI screenshot: panel 4.

Figure 2.10: VS GUI screenshot: panel 5.

Chapter Three

A Fault Tolerant System

Distributed systems are vulnerable to partial failures. By definition, a partial failure is a failure that defects a part or some parts of the system [7]. There are two ways to address this issue. The first one, which is easier, is to consider the partial failure a complete failure and the system has to restart from beginning. The second way is to design the system to tolerate the failure and to continue correctly function. Fault-tolerance needs substantial design and implementation efforts. A lot of techniques, concepts, and solutions have been developed to tackle this issue and other distributed-specific issues, such as, heterogeneity, synchronization, security and process migration.

In chapter 1, we mentioned that we chose Jini, a distributed middleware system, as a steering wheel to ease the implementation and enhance the robustness of our system and this chapter discusses the details.

3.1 Running multiple Lookup Services (LUs)

The LUs represent the keystone for the system since they keep track of VRs and RPs available in the system. To be more specific, the LUs advertise those VRs and RPs at the highest level in the tree (those are not enslaved by other VRs). Then, every VR has the knowledge about its direct children. Since the supervisor can only interact with those VRs and RPs at the highest level in the tree, he relies on the LUs to know what entities are available for interaction.

To increase the system robustness, multiple LUs should be run; each one is on a separate machine and has exactly the same knowledge as others. If one fails, others will backup its functionality.

3.2 Tolerating Virtual Robot's failures

As each VR knows about its direct children and has the handle (see chapter 1) to communicate with them, then what will happen when a VR crashes or its network connection goes down? Will the whole sub-tree it is rooting be lost? Fortunately, the answer is no. This is made possible by adopting the idea of "leases" from Jini, and here is how it works.

When a VR tries to enslave another VR or RP, the two parties agree on lease duration. The enslaving VR says, "I want to enslave you for T time units"; the enslaved VR grants this lease just for this duration. Ironically, when VR1 wishes to enslave VR2, VR1 petitions to VR2 to accept a lease contract. Nevertheless, we will keep using the "enslave" terminology.

While running, the enslaving VR (the lessee) maintains extending the lease for T

periods before the lease expires. The enslaved VR or RP (the lessor), on the other hand, continues granting lease renewals. If it happens that the enslaving VR does not ask for lease extension and the lease duration expires, the enslaved VR or RP will assume that its direct parent (the enslaving VR) has unexpectedly died and has been unable to extend the lease•. Then, the enslaved VR reregisters with LUs to re-announce its existence. The following diagrams explain this process.

I- VR1 asks to enslave VR2 for a period, T. VR2 accepts this and starts a timer that expires after T (Figure 3.1).

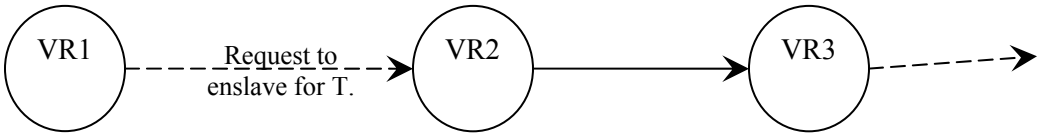

Figure 3.1: At enslaving time.

To ensure no racing condition may occur, VR1 sends lease renewal messages before T elapses by amount of time ε. ε should be sufficient to count for any communication delay the renewing message needs to travel from VR1 to VR2. VR2 does the same thing with VR3 (assuming we have a topology as in figure 3.2)

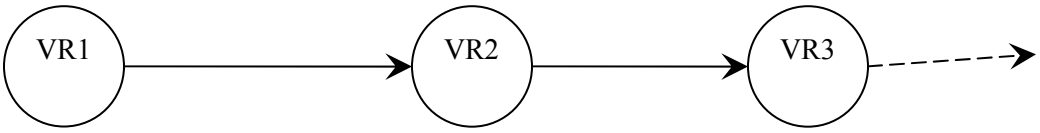

Figure 3.2: Some topology for three VRs.

• This discusses when the parent VR dies unexpectedly. In the case when VS deletes a VR, the deleted VR informs its direct children before it gets disposed.

II- At some time, VR2 suddenly crashes or becomes "networklogically" unreachable (figure 3.3).

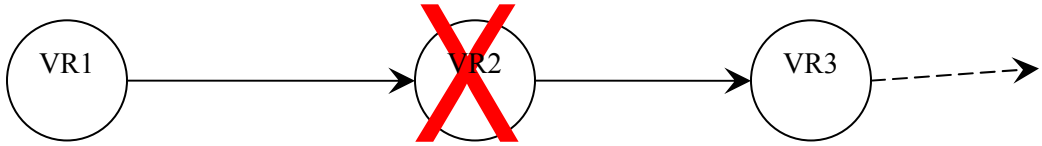

Figure 3.3: VR2 crashes.

VR2 is no longer alive to renew its lease with VR3, the T period passes and VR3 does not hear from VR2. At this point, VR3 registers with LUs. Once VR1 tries to contact VR2 to renew the lease or to execute robotic commands, a communication exception occurs. Then, VR1 releases the pointer (the proxy object) held to VR2. This results in the outcome shown in figure 3.4.

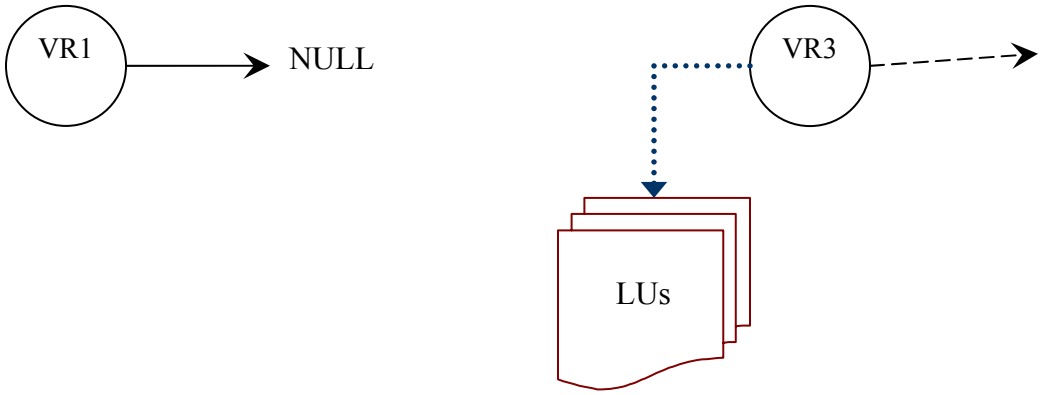

Figure 3.4: The final outcome.

When the supervisor becomes aware of this, he may to re-create a new VR, instruct it to enslave VR3 and bind it to the free (NULL) pointer.

The Lease period (T)

The lease period (T) is an adjustable value. It should reflect the system requirements. There are two contradictory requirements. On one hand, the lease period should be as large as possible. Every two parties engaged in the enslaving relation will exchange renewal messages very frequently it T is small. This will influence the computation time of the VRs and their ability to execute robotic commands. The reason for this is that theses messages have higher priority than robotic command messages since they maintain the VR-tree structure. On other hand, this value should be set to a small value to let the system heal and recover as soon as possible. When a VR unexpectedly crashes, its child waits until the remaining time of the last T to expire before it reregisters with LUs.

So choosing this value should trade off these two requirements. Right now, it is set to two minutes but this is not an optimum value, it is selected to be somehow small for testing purposes.

VR Mobility

When a VR moves, its current state is captured, frozen, serialized, and then written to a stream of bytes to be shipped to the destination. During this, the moving VR is a passive element, i.e., it is not running.

To ensure consistency among system parties, when a VR is moving:

- It is children should pause the lease period (T) timers.
- Its parent is not allowed to send to it messages.
- Neither its parent nor its children are allowed to move.

The moving VR is given a time-out period to reach the destination. If it cannot arrive within time, it will be considered lost and other parties will act accordingly. It may happen that the moving VR just takes time more than the time-out to reach the destination. To ensure no contention will happen in such case, when the time-out expires, the parent releases the pointer held for the moving VR, and the children, which are already registered with LUs, do not accept messages from any VR. The supervisor then decides what to do once the moving VR appears in LUs, e.g., instruct it to re-enslave the children, rebind it to the pointer at the parent, or just dispose it.

The last technique we deployed in our system is the JavaSpaces Technology.

3.3 Deploying the JavaSpaces Technology

JavaSpaces is basically a service registered with Jini LUs. It provides a persistent storage where objects can be written to and later taken or read from.

So far, we have used shared variables written to the space (JavaSpace). Each variable is related to a single robotic command in every VR or RP, for which it keeps track of the number of calls made to that command. If a VR or RP crashes and another copy is started later, this copy will know how many times a call has been made to each of its methods (commands). Until now, there is no great benefit from this but this idea can be matured and made useful for the case when a VR or RP goes down in the middle of a command execution. Then, another copy of same type of the crashed entity can continue from the point a failure has happened. This will lead to a reliable, robust and fault tolerant system.

Chapter Four

The System at Work

This chapter applies our system to ParaDex, one of the robots at Case Western Reserve University[•]. First, we discuss how to write new Virtual Robots to extend the functionality of already existing ones. Next, we demonstrate how to use the Virtual Supervisor to interact with other system entities through a number of screen shots.

4.1 Writing new VRs

As mentioned in chapter 3, the VirtualRobot class is the general class that has the required functionalities to enable a VR to be created, to enslave other VRs or RPs, to be enslaved by another VR, to migrate to another machine in the network, and to be disposed of. Every VR extends the VirtualRobot class to inherit these functionalities and adds new ones. The following example assumes that we have a mobile robot that has one basic command, move. The move command accepts two-real-number arguments and causes the robot to move a distance, in inches, in the X and Y directions equal to the first and the second arguments, respectively. The current location of the robot, once it receives the move command, is defined as (0, 0); positive values mean move forward; negative values mean backward. Given these specifications, one can instruct the robot to move in some geometrical shapes – triangles, squares, pentagons and others. Suppose in some applications moving the robot in square-shape paths is a useful task and needs to be repeated several times. Therefore, we need to automate this task in a single function

[•] For more information about ParaDex, visit http://vorlon.cwru.edu/~vxl11/NetBots//

called “moveSquare”. To be flexible, the moveSquare function accepts a real number argument that specifies the side length of the resultant square. Let us walk through the following code, which accomplishes this function, line by line•.

```
1     import com.ibm.aglet.*;
2     import net.jini.discovery.*;
3     import net.jini.core.lookup.*;
4     import net.jini.core.event.*;
5     import net.jini.core.lease.*;
6     import net.jini.lookup.*;
7     import net.jini.core.entry.*;
8     import net.jini.lookup.entry.*;
9     import java.rmi.*;
10    import java.rmi.server.*;

11    public class moveSquareVR extends VirtualRobot {

12          public void onCreation(Object o) {
13                super.onCreation(o);
14                ROBOTs = new ROBOT[1];
15                ROBOTs[0] = new ROBOT("RobotProxy");
16          }
17          public boolean handleMessage (Message msg) {

18                if(msg.sameKind("moveSquare")) {

19            double L = ((Double) msg.getArg(“length”)).doubleValue();

20                      Message toRopotProxy = new Message (“move”);

21                      toRopotProxy.setArg(“X”, L);
22                      toRopotProxy.setArg(“Y”, 0.0);
23                      ROBOTs[0].sendMessage(toRobotProxy);

24                      toRopotProxy.setArg(“X”, 0.0);
25                      toRopotProxy.setArg(“Y”, L);
26                      ROBOTs[0].sendMessage(toRobotProxy);

27                      toRopotProxy.setArg(“X”, -1*L);
28                      toRopotProxy.setArg(“Y”, 0.0);
29                      ROBOTs[0].sendMessage(toRobotProxy);

30                      toRopotProxy.setArg(“X”, 0.0);
31                      toRopotProxy.setArg(“Y”, -1*L);
32                      ROBOTs[0].sendMessage(toRobotProxy);

33                      return true;
```

• The examples shown here are simplified in that they do not have the exception handling statements required when sending messages through the network.

```
34                }

35                else if(msg.sameKind("move")) {
36                      ROBOTs[0].sendMessage(msg);
37                      return true;
38                }
39                else return super.handleMessage(msg);
40          }
41    }
```

Example 4.1: Example of how to write a VR.

Lines 1-10 import the class libraries (RMI, Jini, and Aglet libraries) required to compile this Java program. Line 11 declares that the new VR to be of type moveSquareVR and it extends the VirtualRobot class. The moveSquareVR overrides the two methods onCreation (lines 12-16) and handleMessage (lines 17-32) inherited from the VirtualRobot class as explained next. When the VirtualSupervisor creates any VR, it gives the VR a name, as shown in the next section. This name is passed to the base class (line 13), which is the VirtualRobot, to initialize the NAME member data. The name is useful to differentiate multiple VRs if they are of same type. Lines 14 and 15 show how to define children to a VR. The number of children (line 14) and each child's type (line 15) are defined – in this case, there is just one child of type "RobotProxy".

In the handleMessage method, the VR expects to accept three categories of messages. First, a message of kind "moveSquare" (lines 18-34), which is the new functionality introduced by this VR. Second, a message of kind "move" (lines 35-38), which exposes the functionality of the RobotProxy enslaved by this VR. If the VR does

not explicitly expose this functionality from the downstream RP, entities interacting with this VR will be unable to use and call the “move” task. Therefore, this VR has a blend of basic low-level commands and compound commands. Third, control messages (line 39), which are used to maintain leases with parent and children, to instruct the VR to migrate to another machine, to enslave other VRs, and the others mentioned in chapter 2. The implementations of control messages are in the VirtualRobot class.

Line 19 extracts the side length argument of the square from the incoming message (refer to the aglets API [16] for a complete discussion about creating aglet messages, setting arguments, and retrieving the arguments from a message). Line 20 creates an aglet Message of type “move” to be shipped to the child. Lines 21, 22, and 23 set the X and Y arguments of the message, and send the message to the child. It is assumed that the child (the RP) has the following piece of code.

```
public boolean handleMessage (Message msg) {

      if(msg.sameKind("move")) {

            double Xdirection =((Double)msg.getArg(“X”)).doubleValue();
            double Ydirection =((Double)msg.getArg(“Y”)).doubleValue();

            // send these values (Xdirection and Ydirection) through
            //the socket connection to the RPCS.

            return true;
      }

      // other else if statements ……
}
```

Example 4.2: Part of the RP’s handleMessage method.

The four-set of statements (lines 21-23, 24-26, 27-29, and 30-32) set the appropriate arguments and send the messages to the child. These statements cause the robot to move in a square-shape path like the following diagram starting at (0, 0).

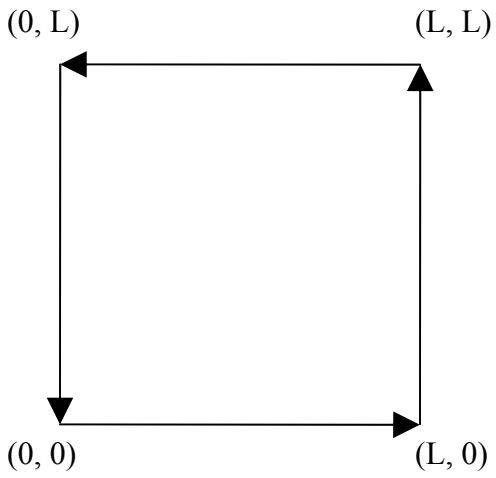

Figure 4.1: Moving in square-shape path.

If the message is of kind “move”, the VR sends the message as is to the child (line 36). Otherwise, the message is passed to the base class, the VirtualRobot class, to be processed (line 39).

After writing a VR, the TaskGUI is written to allow the VirtualSupervisor to interact with the VR in a convenient way. Inside the TaskGUI class, the moveSquare may be called as follows:

```
1     msg = new Message("moveSquare");
2     msg.setArg("length", new Double(y.getText()));
3     proxy.sendMessage(msg);
```

Example 4.3: Calling the moveSquare method.

Line 1 creates a message of kind moveSquare. Line 2, which assumes that the GUI has a text field to input the length of the square side, sets the value of the length argument. Line 3 sends the message to the VR by calling the sendMessage method on the corresponding proxy object.

The next section shows how to create VRs, enslave them, and interact with them through the VS GUI panels.

4.2 A Running Scenario

We have tailored the RobotProxy to fit the functionality of the ParaDex RPCS. The following are the major low-level commands of the ParaDex:

- *goTo:* this method causes the robot arm to move to a point in the three-dimensional space and to rotate the gripper in a particular angle.
- *getInfo:* this method reports the current location, the twist angle of the robot arm, and the forces exerted by the robot arm in all direction.
- *gripper:* provided with a single parameter, this method closes or opens the gripper attached to the arm.

The following screen shots demonstrate how to use these basic commands to build high-level and more useful commands.

The first screen shot (figure 4.2) shows when the VS is started using the aglet viewer [16 and 26]. At this point, both the ParaDex RP and WorkSpaceView are running in the system.

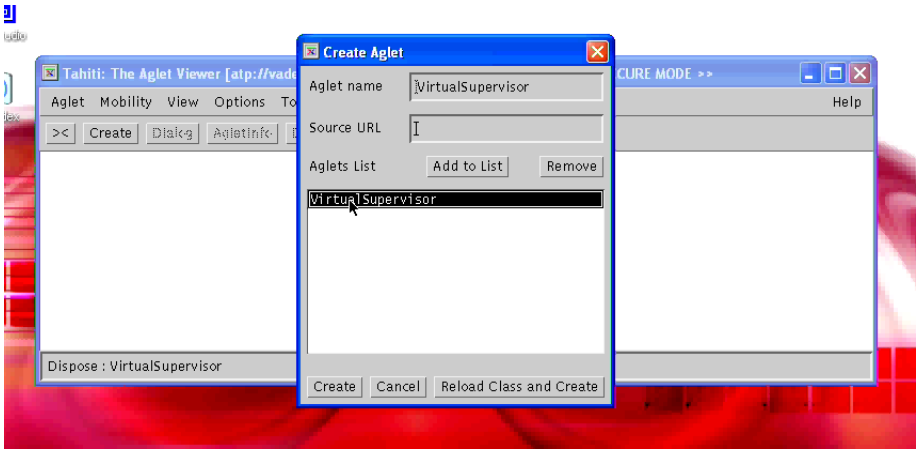

Figure 4.2: Starting VS.

Selecting the VirtualSupervisor in the previous window and pressing the create button launches the VS GUI. Once started, the VS contacts the LUs and gets the RP and the WorkSpaceView (figure 4.3).

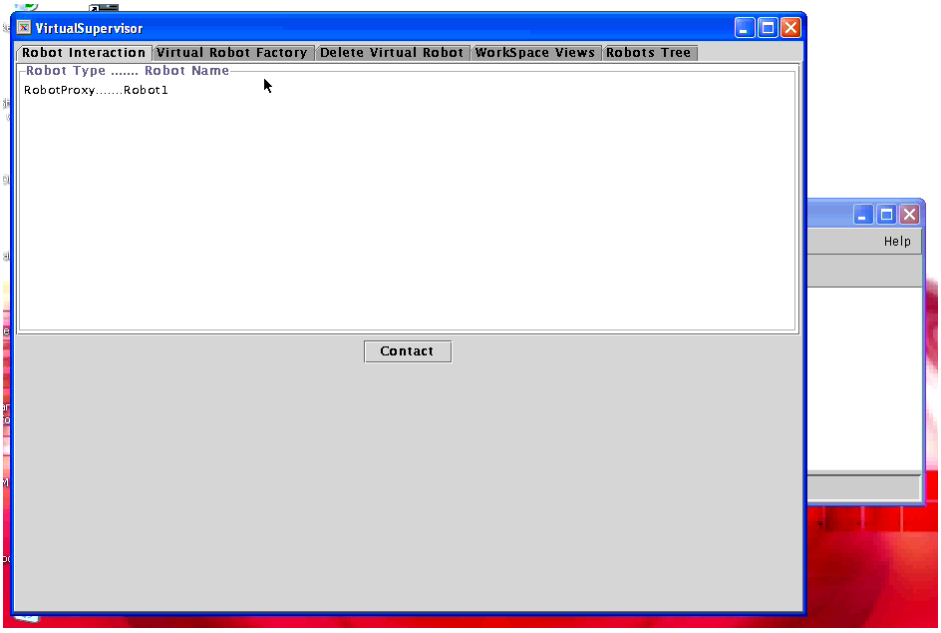

Figure 4.3: The VS GUI.

After the user writes a VR, as shown in the previous section, he needs to compile the Java source file, which gives a class file. Figures 4.4 and 4.5 show how to instantiate and create the VR. After choosing the VR class file, a window pops up that prompts the user to enter the agent's (VR) name. When instantiated, the VR registers with LUs and the LUs notify the VS.

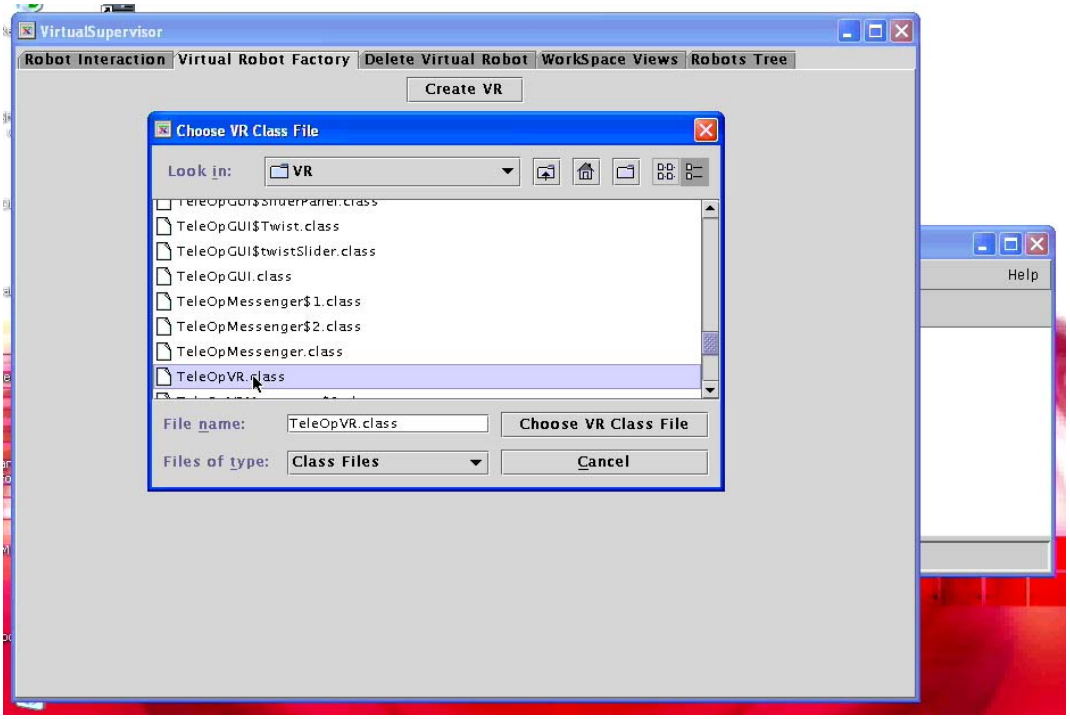

Figure 4.4: Creating the TeleOpVR VirtualRobot, selecting the class file.

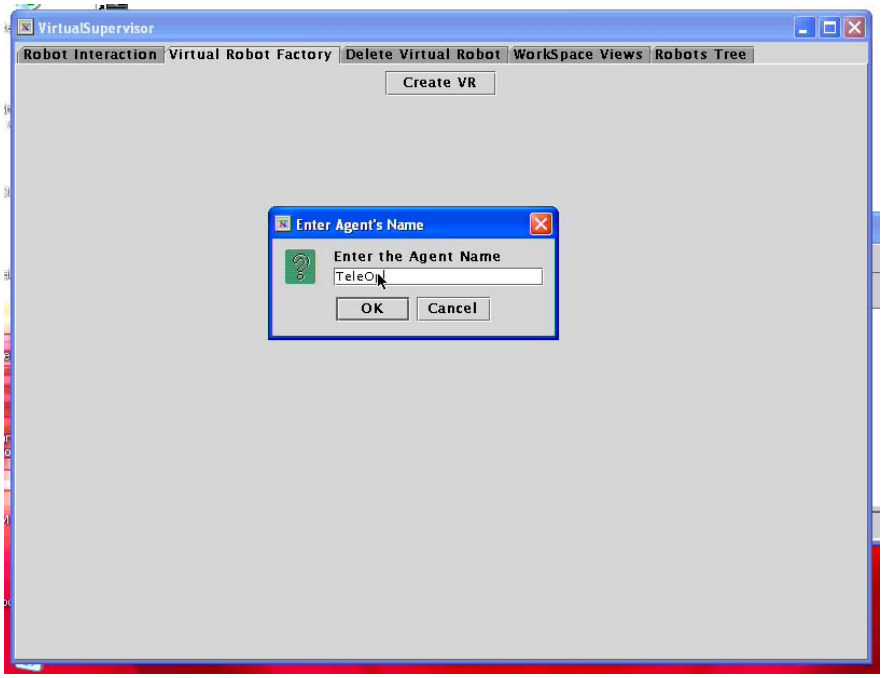

Figure 4.5: Creating the TeleOpVR VirtualRobot, naming the agent.

The next step is to have the VR enslave the RP (figure 4.6). As shown in the pervious section, the VR specifies the type of its children but does not specify their names. The VS allows the user to choose among VRs those match the specific type but have different names. Once enslaved, The RobotProxy is hidden (figure 4.7). Figure 4.7 also shows the user requesting the TeleOpVR TaskGUI whereby a GUI window pops up on the screen (figure 4.8).

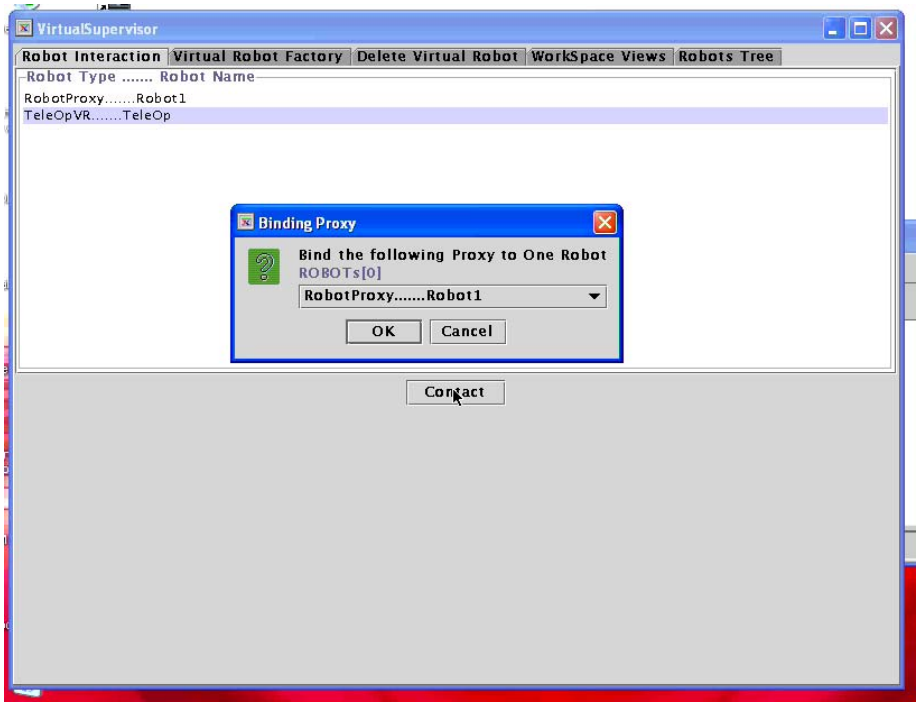

Figure 4.6: Using TeleOpVR to enslave the RobotProxy.

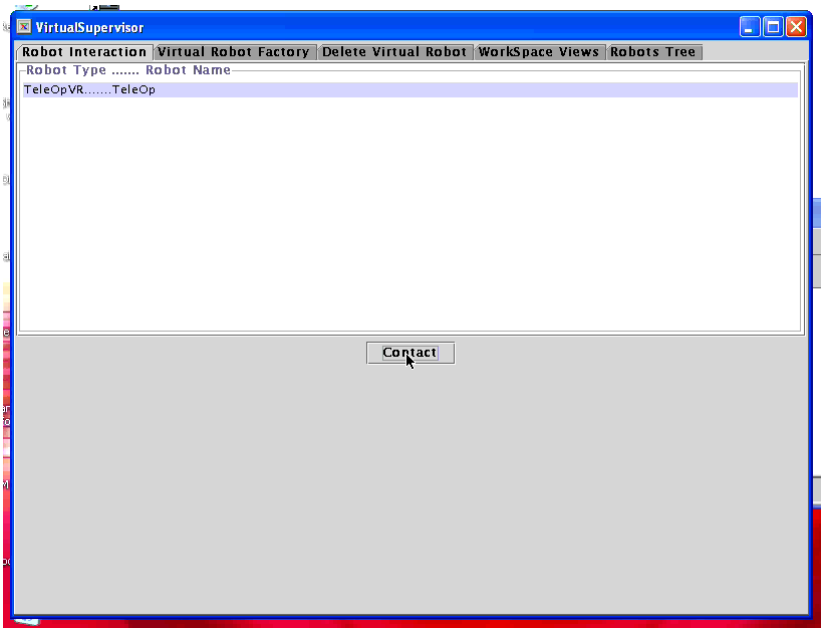

Figure 4.7: Requesting the TeleOpVR TaskGUI.

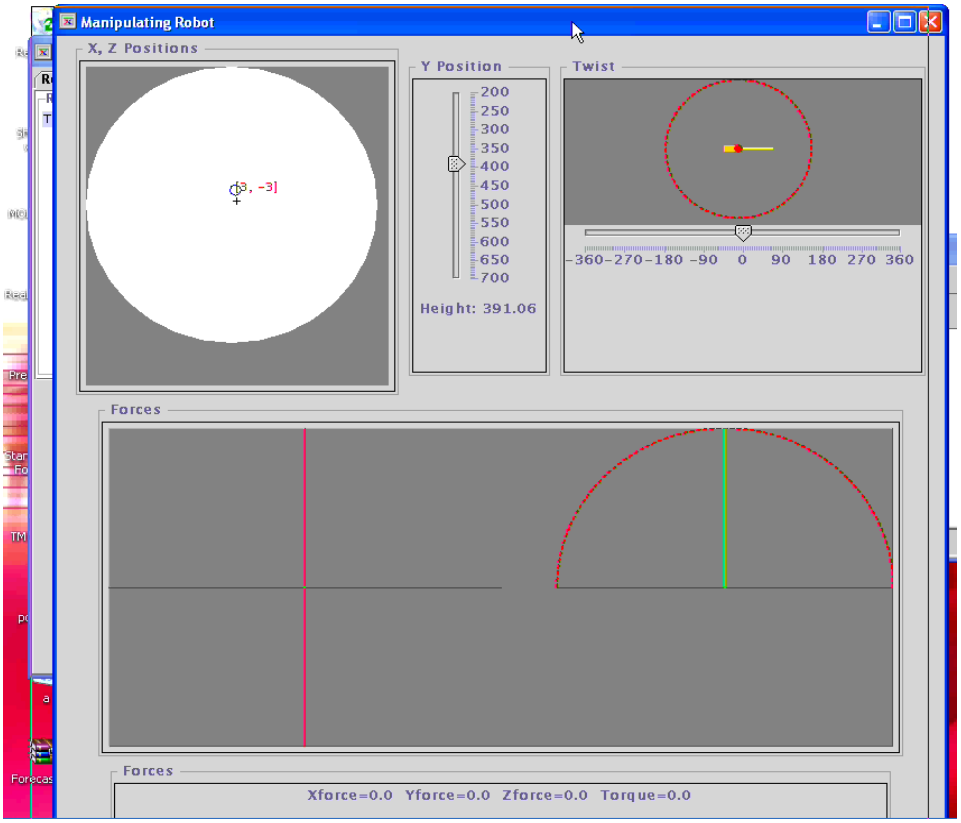

Figure 4.8: TeleOpVR TaskGUI.

The upper GUI components in figure 4.8 enable the user to instruct the robot arm to move in the X, Z, and Y directions and to rotate the gripper certain degrees in both directions. Clicking with the right mouse button inside the white circle alternatively closes and opens the gripper. The other GUI components show the feedback forces exerted by the robot arm in all directions in graphical and textual forms.

Figure 4.9 shows when the user requests to view the robot's workspace, a window then pops up showing the workspace view (figure 4.10).

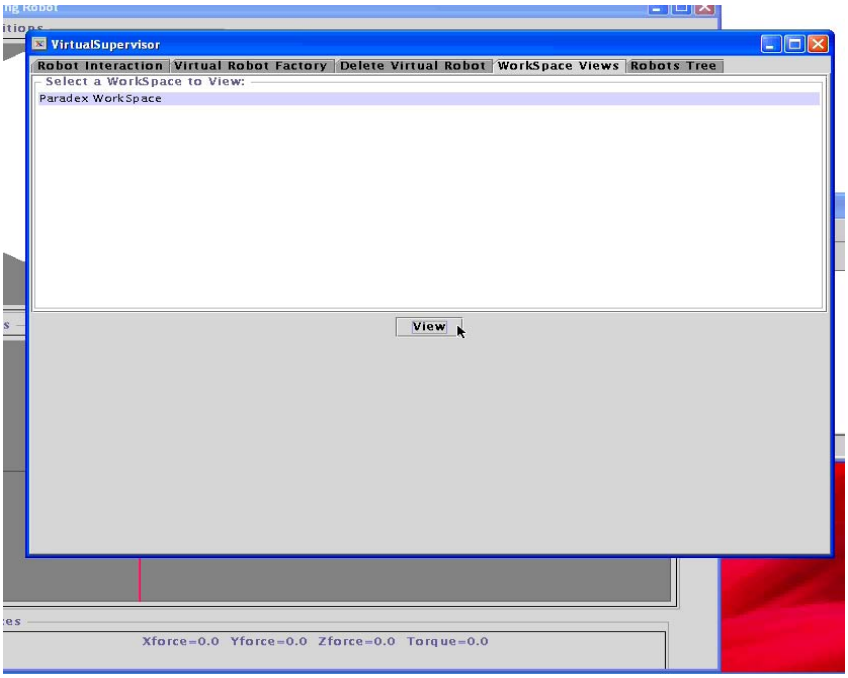

Figure 4.9: Viewing the workspace.

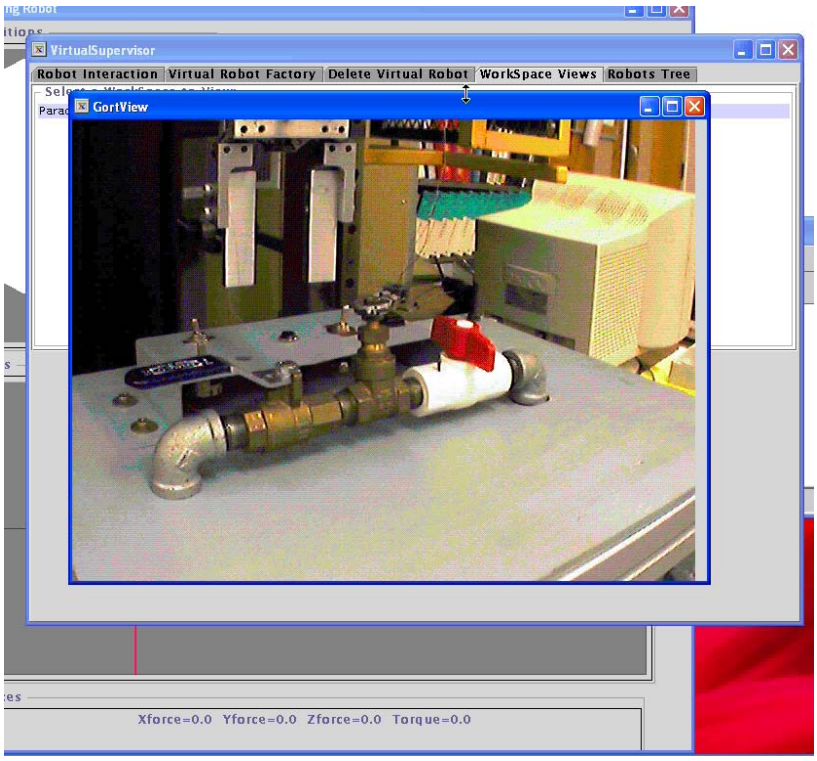

Figure 4.10: ParaDex Workspace View.

Figures 4.11 to 4.14 show how the user manipulates an object in the robot's workspace, closing a lever, in tele-operation mode.

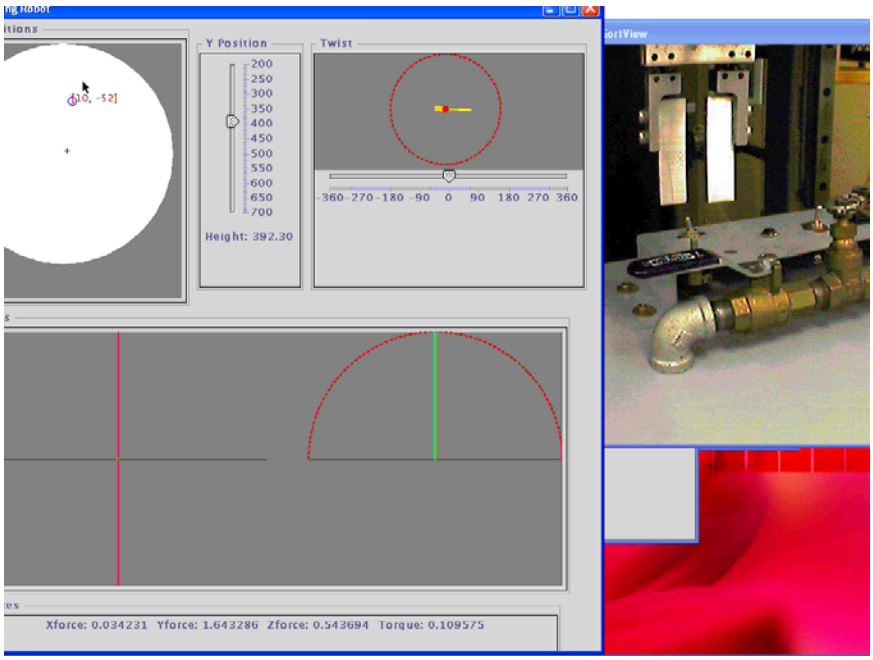

Figure 4.11: Moving in the X-Z (horizontal) plane.

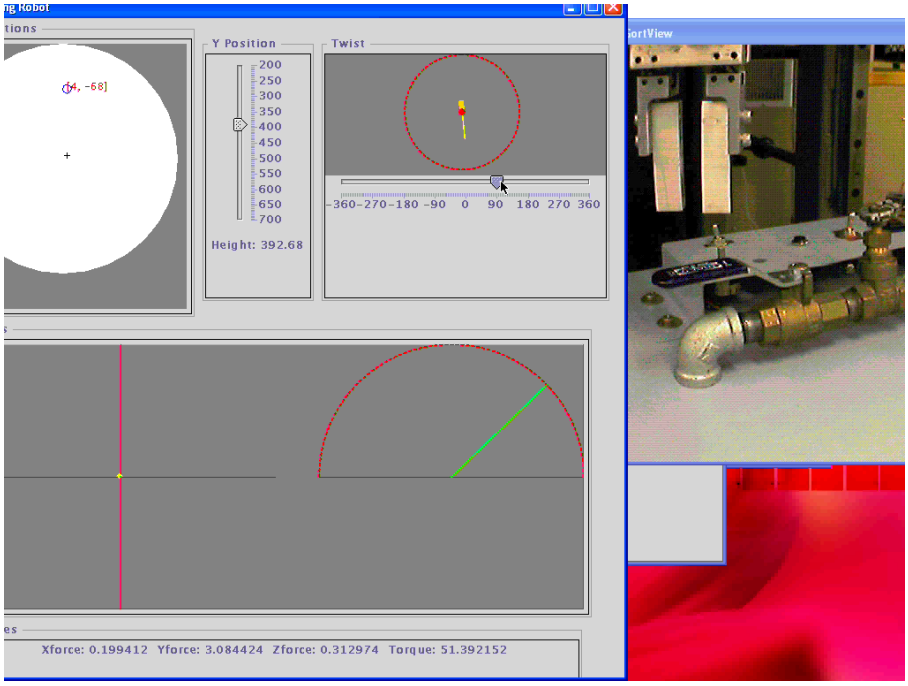

Figure 4.12: Twisting the gripper.

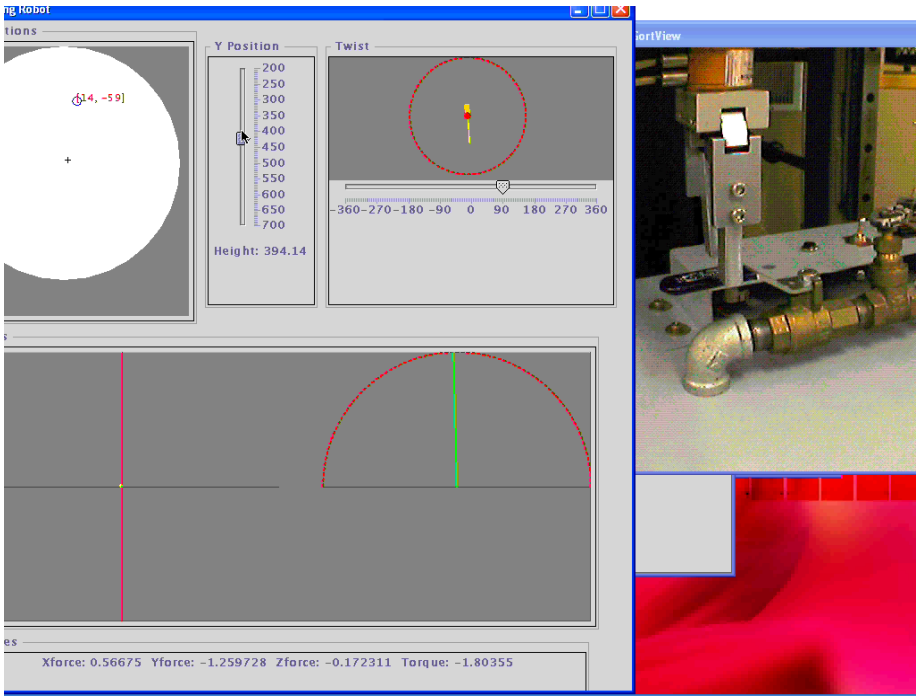

Figure 4.13: Descending to catch the gripper.

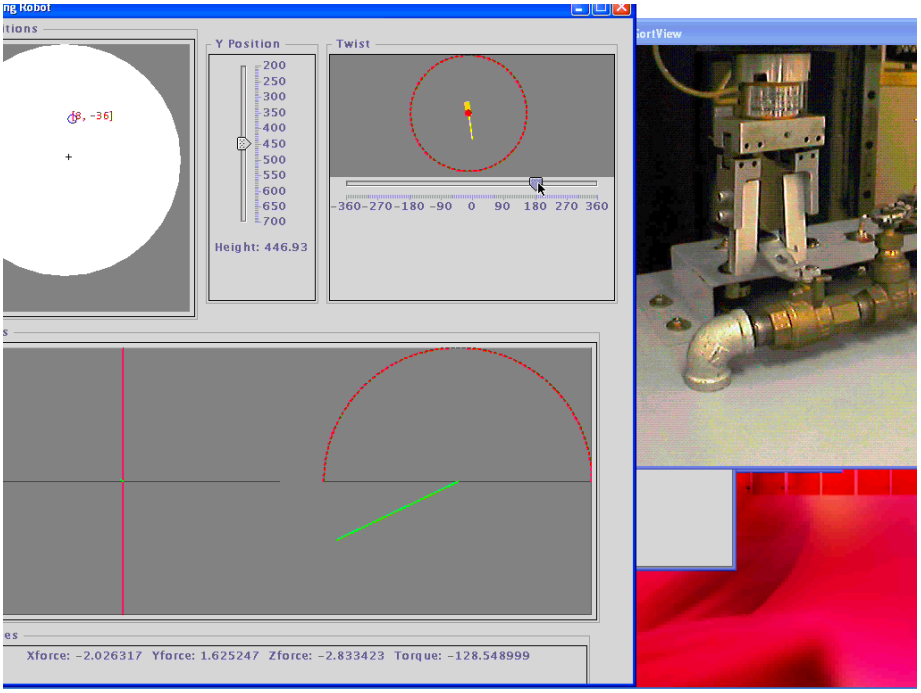

Figure 4.14: Closing the lever.

Now, the user wishes to automate the close lever task. He writes two VRs: one to support a move command and another to support the close lever command•. The VR that supports the move command exposes other low-level commands to its parent, which are getInfo, gripper, and goTo.

When the user deletes the TeleOpVR (figure 4.15), the RobotProxy re-registers with LUs and the VS becomes aware of this (figure 4.16).

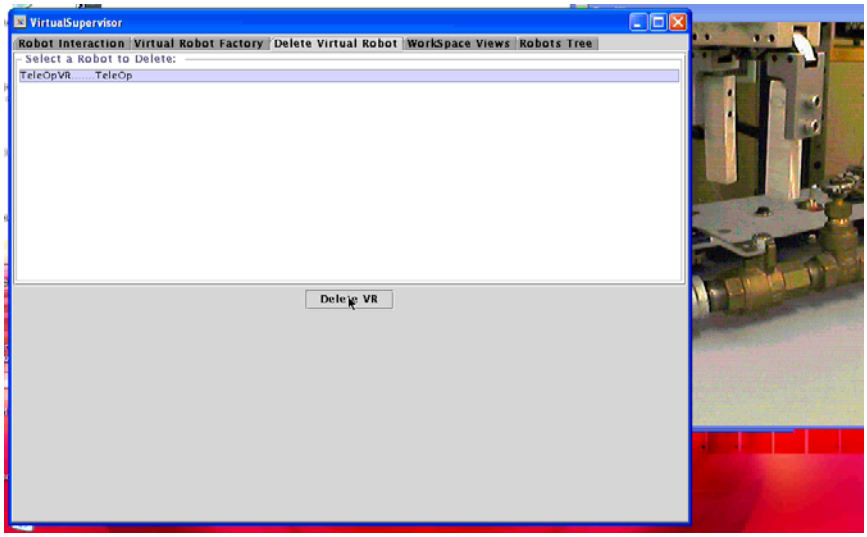

Figure 4.15: Deleting the TeleOpVR.

• The move command is different from the goTo command. The goTo command causes the robot arm to travel to the destination in one direct and jerky movement, whereas the move command causes the arm to travel to the destination in small steps and in a smooth movement. The move command consists of several goTo and getInfo commands executed in sequence.

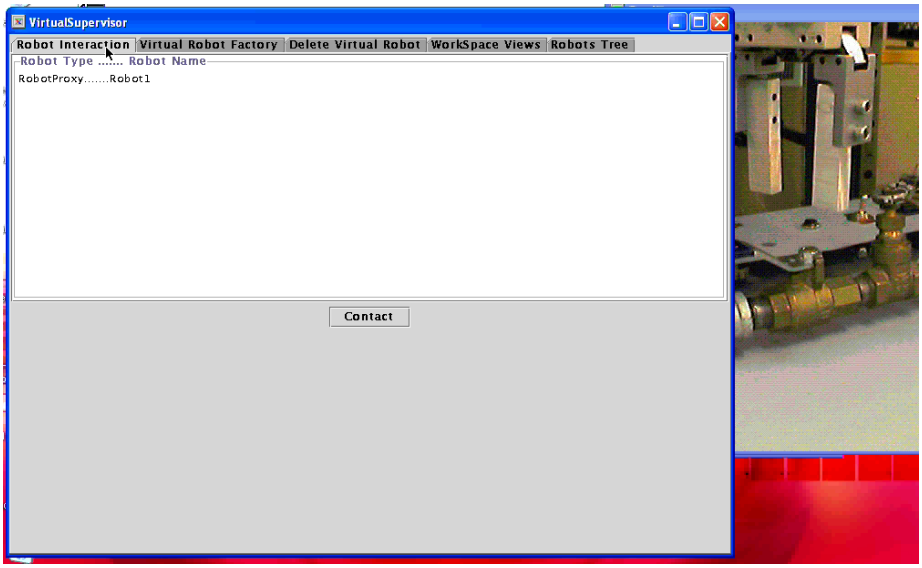

Figure 4.16: The RobotProxy re-registers with LUs.

Then, the user creates the two new VRs, MoveToVR and OpenCloseLeverVR, instructs MoveToVR to enslave the RobotProxy, and instructs OpenCloseLeverVR to enslave MoveToVR (figures 4.17 – 4.22).

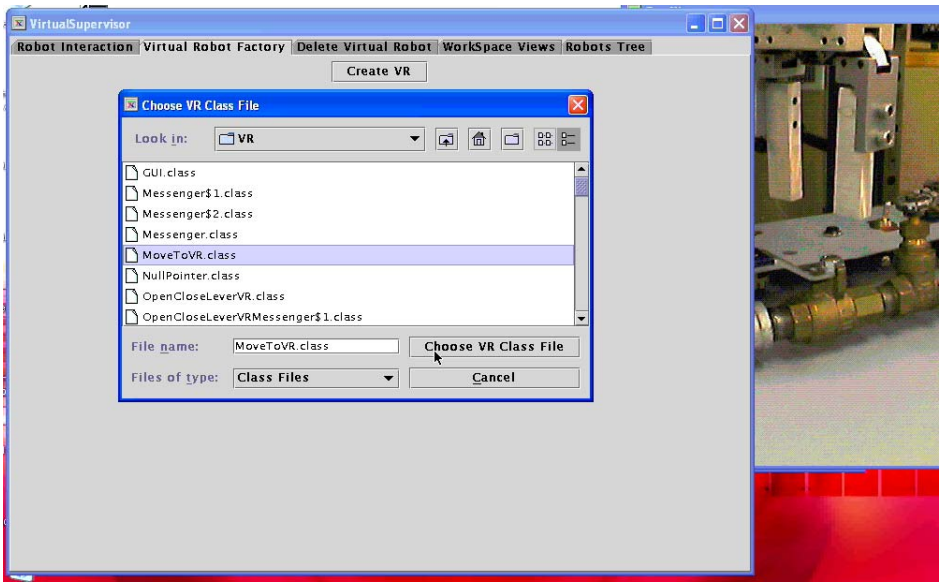

Figure 4.17: Creating MoveToVR.

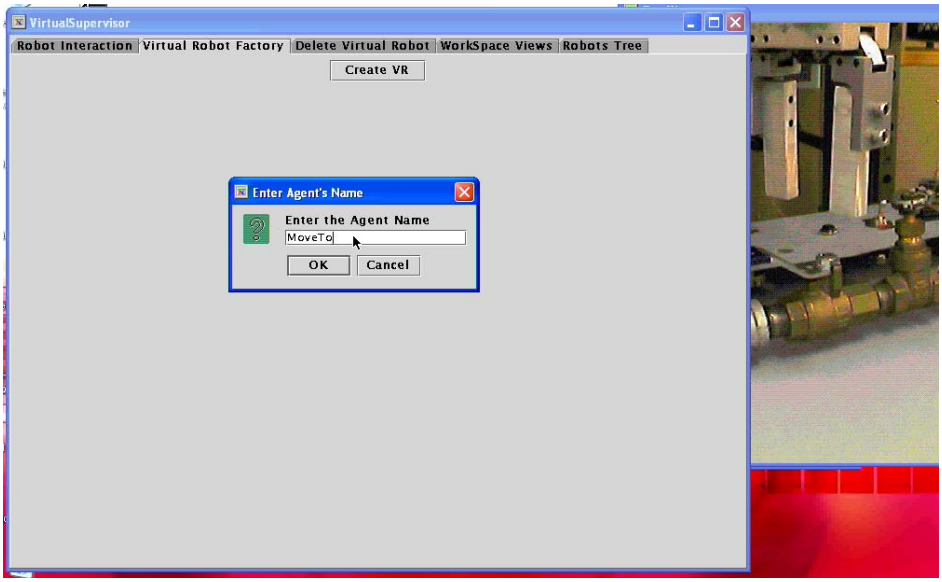

Figure 4.18: Entering the agent's name.

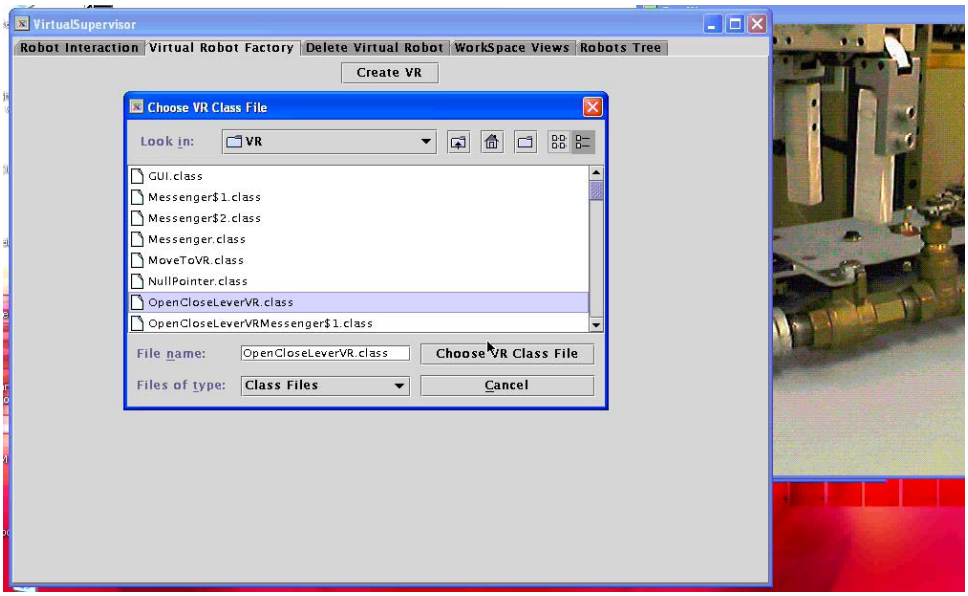

Figure 4.19: Creating OpenCloseLeverVR.

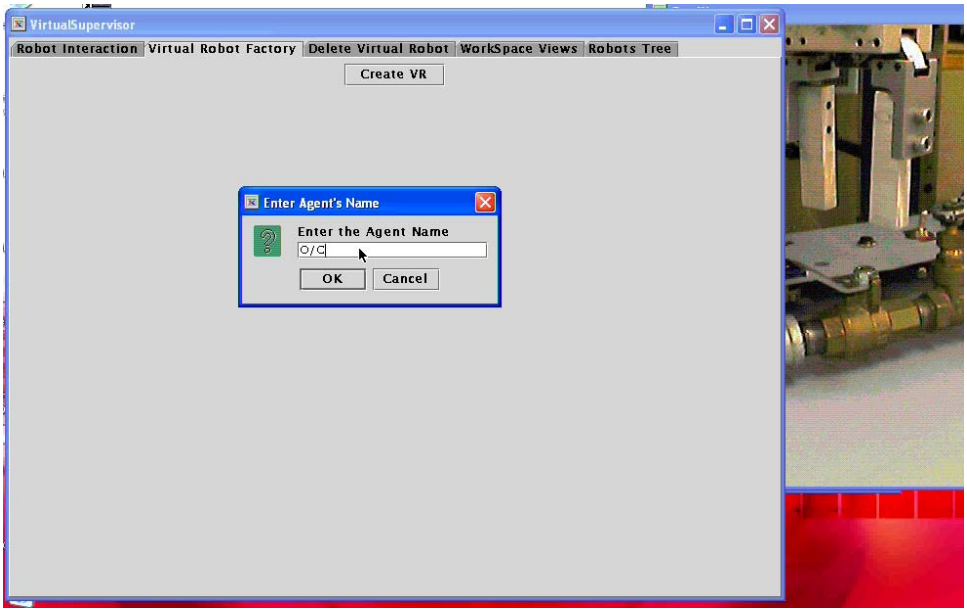

Figure 4.20: Entering the agent's name.

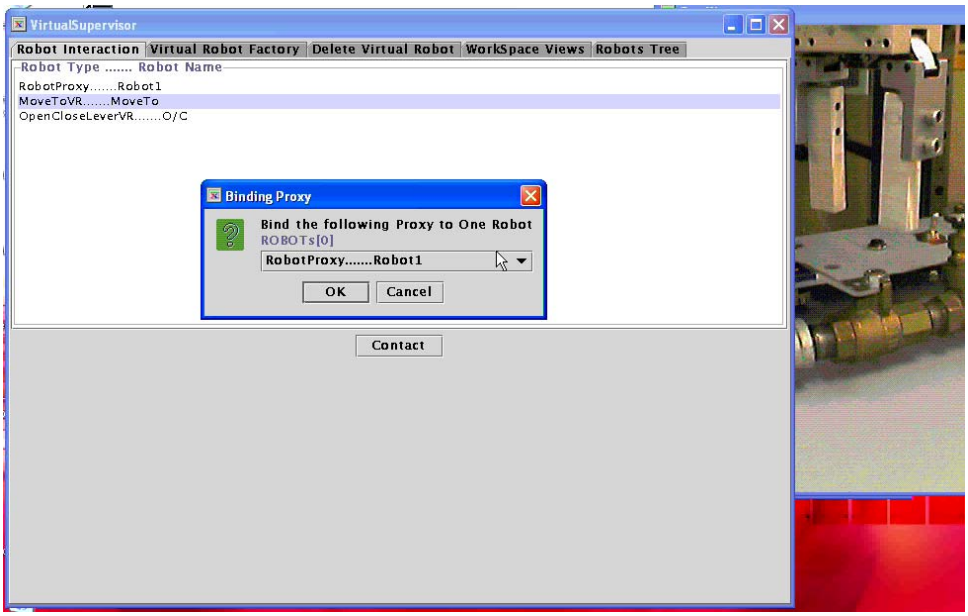

Figure 4.21: Enslaving the RobotProxy.

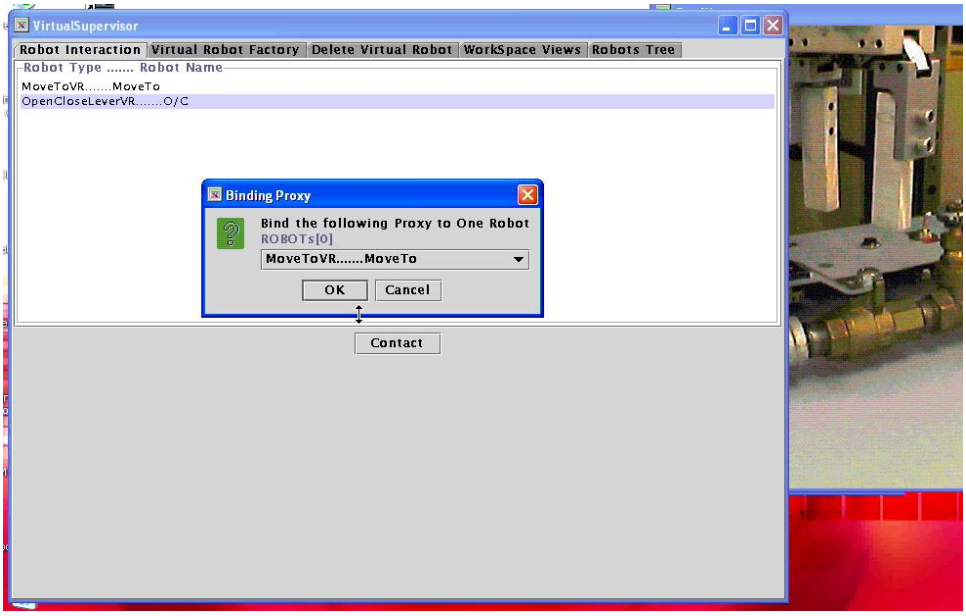

Figure 4.22: Enslaving the MoveToVR.

After requesting OpenCloseLeverVR's TaskGUI (figure 4.23), the user needs only to press a button in the GUI to close the lever (figures 4.24 and 4.25). The user can also use the same VR to open the lever.

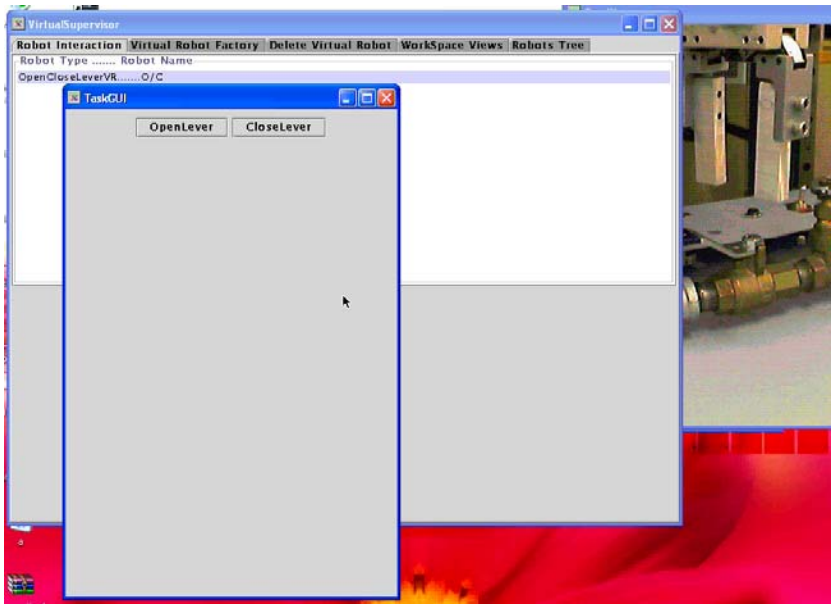

Figure 4.23: Requesting the OpenCloseLeverVR TaskGUI.

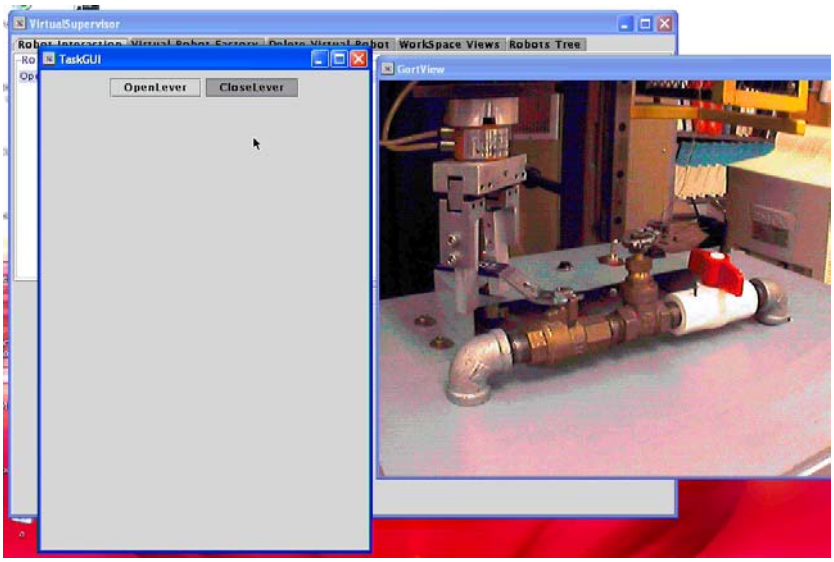

Figure 4.24: Pressing the CloseLever button.

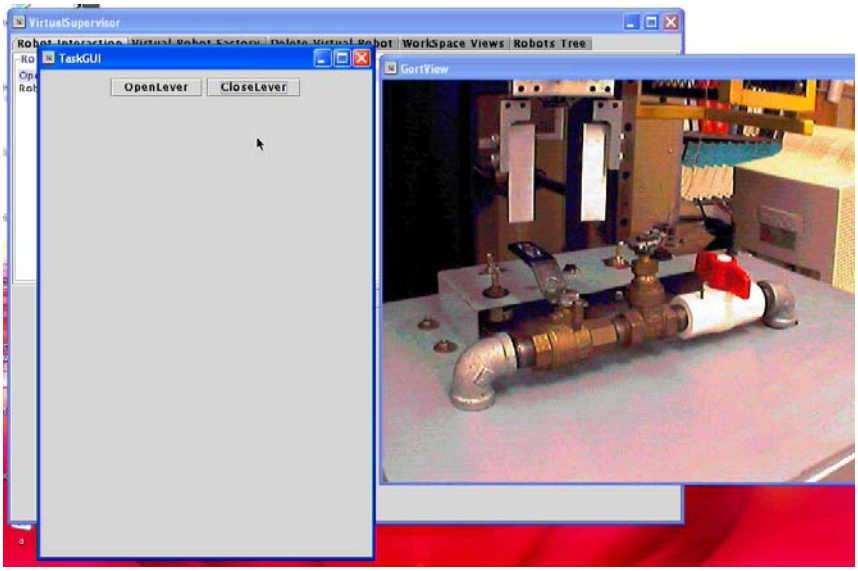

Figure 4.25: The robot's arm accomplished the close lever task.

The example above demonstrates how to use the system to extend the functionality of a robot based on a basic predefined set of tasks the robot supports in a very similar way a software developer uses the language's library functions to implement useful functions specific to his needs. As the robot specific language and the real-time constraints are encapsulated within the RPCS and the controller, the person who wants to use this system needs not to be a "robotist" (robot scientist). Indeed, section 1 of this chapter showed that writing a code to a robot is just writing an ordinary Java program.

Chapter Five

Summary and Future Work

This chapter concludes the thesis by presenting the summary and directions for future work.

5.1 Summary

We have implemented a distributed system to facilitate human-robot interaction. The main advantage of the resulting system is the expandability and on-the-fly programmability provided to the user, who is even not mandated to be a robotist. Therefore, we have gone beyond static human-robot interaction where the user needs to adhere to a fixed number of functionalities a robot supports. This flexibility is made possible by using compliant control robots where the real-time constraints are encapsulated within the robot controller.

Throughout system design and implementation stages, the intent was to build a generic system that can fit a large number of robots. In addition, the verification tests used were based on general scenarios and not based on any particular robot. The system has nothing specific to robots and can be used to interact with any intelligent system remotely. When the system parts were completed, we applied the system as a whole to an arbitrary robot as shown in the previous chapter. In this example, a robot arm is used to open and close a lever, which demonstrates the ability of the system to work in physical, real world environments.

To implement individual parts of the system, we relied on existing, well-established techniques whenever possible. Thus, we used the Aglet mobile agents to provide the code with the ability of migration from one machine to another in a network, and Jini to provide the lookup directories, lease and discovery utilities.

Our system also allows the user to control multiple robots using one control panel, although this capability has not been yet tested using physical robots.

The next section discusses potential ideas that can be augmented to the current system to produce more enhanced releases.

5.2 Future Work

As mentioned in chapter 3, we have used JavaSpaces, as a basic idea, to keep track of the number of calls made to each method in a VR. If this VR crashes and another copy is started afterward, the second copy can know how many times a call has been previously made to any of its methods. More work should be conducted to have the second copy of the VR continue from the point where the crashed VR has stopped. Issues to be considered are whether to instantiate many copies of each entity at once, or to instantiate only one copy and when something later happens, another copy is instantiated; and how to address the problem when a VR crashes in the middle of method execution.

A second idea for future work is to automate the Virtual Robots mobility. Right now, the user instructs VRs to move from one machine to another. As each VR's demands of resources, such as memory, processing power or network quality; and time constraints may change dynamically, there is a need for the VR to measure such metrics and to decide by its own when and where to migrate. VR mobility can also help to

compromise between distance and autonomy. VRs that need to exchange low-level commands and need to fulfill time constraints will move close to each other, whereas VRs that are far away from each other should exchange high-level commands that do not need time constraints.

Finally, as a matter of perfection and providing easiness, there is a need to automate code writing for VRs and corresponding TaskGUIs. For example, providing the user with interface where he can specify the functionality of a VR using a graphic notation while keeping code writing at minimum. Then, this interface automatically generates the Java source code.

Bibliography

[1] Bottazzi, S., Caselli, S., Reggiani, M., & Amoretti, M. (2000, October). A Software Framework based on Real-Time CORBA for Telerobotic Systems. *In IEEE/RSJ International Conference on Intelligent Robots and Systems*.

[2] Box, D. *et al*. (2000, May). Simple Object Access Protocol (SOAP) 1.1. *W3C.org*. Retreived November 6, 2003, from http://www.w3.org/TR/SOAP/

[3] Chopra, N., Spong, M. W., Hirche, S., & Buss, M. (2003, June). Bilateral Teleoperation over the Internet: the Time Varying Delay Problem. *In Proceedings of the American Control Conference.*

[4] Dalton, B. *A Distributed Framework for Internet Telerobotics*. Retreived November 6, 2003, from http://www.mech.uwa.edu.au/jpt/tele/MITChapterBJAD.pdf

[5] Dohring, M., & Newman, W. (2002). Admittance Enhancement in Force Feedback of Dynamic Systems. *In Proceedings of IEEE International Conference on Robotics and Automation (ICRA '02), 1*, 638 -643.

[6] Edwards, W. K. (2000). *Core Jini* (2nd. ed.). New York: Prentice Hall PTR.

[7] Freeman, E., Hupfer, S., & Arnold, K. (1999). *JavaSpaces Principles, Patterns, and Practice* (3rd. ed.). New York: Adison-Wesley.

[8] Ghiasi, S., Seidl, M., & Zorn, B. (1999, September). A Generic Web-based Teleoperations Architecture: Details and Experience. *In Proceedings of SPIE Telemanipulator and Telepresence Technologies VI.*

[9] Gill, C. *et al*. (2003, January). ORB Middleware Evolution for Networked Embbeded Systems. *In the Eiththh IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2003)*.

[10] Glosser, G.D., & Newman, W.S. (1994, May). The Implementation of a Natural Admittance Controller on an Industrial Manipulator. *In Proceedings of IEEE International Conference on Robotics and Automation, 2*, 1209 -1215.

[11] Goldberg, K. *et al*. (1995). Desktop tele-operation via the world wide. *In Proceedings of the IEEE International Conference on Robotics and Automation.*

[12] Goldberg, K, & Santarromana, J. (1996). *The Telegarden*. Retreived November 6, 2003, from http://cwis.usc.edu/dept/garden/

[13] Ho, T. T., & Zhang, H. (1999, May). Internet-Based Tele-Manipulation. *In Proceedings of the 1999 IEEE Canadian Conference on Electrical and Computer Engineering*.

[14] Kumaran, I. (2001). *Jini Technology : An Overview*. Prentice Hall.

[15] Lane, J. C. *et al* (2002, November). Effects of Time Delay on Telerobotic Control of Neutral Buoyancy Vehicles. *In IEEE International Conference on Robotics and Automation*.

[16] Lange, D. B., & Oshima, M. (1998). *Programming and Deploying Java Mobile Agents with Aglets*. New York: Addison-Wesley.

[17] Marin, R., Sanz, P. J., & del Pobil, A. P. (2001). *A Complete System Design to Teleoperate a Robot via Web in a Friendly manner: A Real Case*. Retreived November 6, 2003, from http://ciclop.act.uji.es/rmarin/teaching/icra2001_uji_rmarin.pdf

[18] Moody, S. A. (2003, May). Challenges in Building Scalable Network Centric Real-Time Information Dissemination Systems. *In Sixth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'03).*

[19] Ngai, L., Newman, W. S., & Liberatore, V. (2002). An Experiment in Internet-Based, Human-Assisted Robotics. *In 2002 IEEE International Conference on Robotics and Automation (ICRA 2002), 2*, 2190-2195.

[20] Oak, S., & Wong, H. (2000). *Jini in a Nutshell*. New York: O'Reilly & Associate, Inc.

[21] Rosas, D., Covitch, A., Kose, M., Liberatore, V., & Newman, W. S. (2003, Januray). Compliant Control and Software Agents for Internet Robotics**.** *In the Eithth IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2003)***.**

[22] Schmidt, D., & Kuhns, F. (2000, June). An Overview of the Real-Time CORBA Specification. *IEEE Computer Magazine, (33)6*, 56-63.

[23] Shoffner, M. (1998, May). Write your own MOM! Write your own general-purpose, message-oriented middleware. *JavaWorld.com*. Retreived November 6, 2003, from http://www.javaworld.com/javaworld/jw-05-1998/jw-05-step.html

[24] Simmons, R., Fernandez, J., Goodwin, R., Koenig, S., & O'Sullivan, J. (1999). Xavier: An Autonomous Mobile Robot on the Web. *Robotics and Automation Magazine, 7(2).*

[25] Taylor, K. *et al*. (1998, July). *Australia's Telerobot on the Web*. Retreived November 6, 2003, from http://telerobot.mech.uwa.edu.au/

[26] *Aglet Software Development Kit: Summary.* (2002). SourceForge.net. Retreived November 6, 2003, from http://sourceforge.net/projects/aglets/

[27] *CORBA Overview*. Objective Interface**.** Retreived November 6, 2003, from http://www.ois.com/resources/corb-2.asp

[28] *Jini Network Technology*. (2001). Sun.com. Retreived November 6, 2003, from http://wwws.sun.com/software/jini/

[29] *Mars Arctic Research Station*. (2000). Meditac.com. Retreived November 6, 2003, from http://www.meditac.com/mars/overview.html

[30] *Real-Time CORBA Overview*. Objective Interface**.** Retreived November 6, 2003, from http://www.ois.com/resources/corb-3.asp