A Multi-robot-Multi-operator Collaborative Virtual Environment

Moisés Alencastre-Miranda, Lourdes Muñoz-Gómez, Carlos Nieto-Granda, Isaac Rudomin, and Ricardo Swain-Oropeza

Tecnológico de Monterrey - Campus Estado de México (ITESM-CEM), Mechatronics Research Center (CIMe), Atizapan, Estado de México, México
{malencastre, lmunoz, A00461680, rudomin, rswain}@itesm.mx

1 Introduction

In this chapter, we describe the design of an object oriented and distributed architecture and the development of a modular networked system that allow us to have a multi-robot-multi-operator system. On that system, users can collaborate on different robotics applications (e.g. teleoperation, planning, off-line programming, mapping, etc) using robots of different types (e.g. manipulator arms, wheeled mobile robots, legged robots, etc). For this chapter, not only will be addressed the design details in the architecture but also implementation details in the system. Also, is presented an extensive literature and concepts about the multi-user robotics systems.

There are several applications for such systems. In industry, operators must frequently program off-line several manipulators in a workcell. In research, researchers in different cities or countries want to test a planning method with several wheeled mobile robots. In education, students must execute a practice session with several virtual and real robots in a robotics course; etc. For all these cases, the development of software that allows several users to interact in a shared VE with many virtual robots and also communicate with the real robots is very useful.

Multi-robot-multi-operator interaction in a shared VE is a n-to-m relation where there are n users or operators that interact with m virtual robots (n and m are natural numbers). Each virtual robot can control a corresponding real robot of the same type.

A virtual robot is only the visual simulation of a real robot. This means that the virtual robot only has the appearance of the real one. But, the functionality and the features of the real robot must be simulated considering for example the kinematics and dynamics of the robot in order to make a virtual robot move and work inside the VE in a way that is similar to a real robot. However, it is so difficult and computationally expensive to model the exact

D. Brugali (Ed.): Software Eng. for Experimental Robotics. STAR 30, pp. 431–458, 2007.

functionality and features of the real robot in the virtual one. In order to integrate the virtual robots to a VE software is necessary to have the geometric model of each robot in some common format (e.g. OBJ, VRML, etc) and a way to load the models from a program (e.g. in C++, Java, etc). These virtual robots and VE features are the basic elements to develop a Collaborative Virtual Environment for robotics explained in next section.

2 Collaborative Virtual Environments

A Collaborative Virtual Environment (CVE) or Networked Virtual Environment (Net-VE) is a system that allows multiple users in different geographical locations to interact, share, communicate and collaborate in a given task at the same time in a common virtual environment [SZ99] [CPMTT99]. A CVE has the following features:

- A shared sense of space. All participants (users) have the feeling of being located in the same place (inside of a shared VE). The shared space must have the same visual features for all participants; however, each one can see it from a different point of view.
- A shared sense of presence. Each participant can be identified by using a graphical representation in the VE. This representation can be an avatar (digital character animated) or a Graphical User Interface (GUI) showing who is connected to the system at every moment.
- A shared sense of time. Participants must see another participant behaviors or actions in the moment in which these behaviors are taking place.
- A communication mechanism. Because of the participants' remote locations, there must be voice or text communication existing between them (e.g. chat).
- A sharing mechanism. The importance of a CVE lies in the ability of the participants to interact with other participants in the VE, and maybe exchange files.

In other words, a CVE is a multiuser 3D VE system where: all users are running the same software on its computer, everybody are working on the same scenario or environment (VE) that contains the same elements (virtual objects) in the same positions and orientations at any time, one or more group of users can be collaborating in one or more tasks in the VE, but each user can observe a desired area of the VE from a different point of view and zoom configuration. CVEs had been mainly used for games and military training. Four well known CVEs architectures were developed as militar simulation systems during the 80's and 90's decades by the Department of Defense (DoD) and the Naval Postgraduate School (NPS) (e.g.[MZP99], [SZ99], [KWJ99]): Simulator Networking (SIMNET), Distributed Interactive Simulation (DIS), Naval Postgraduate School Network (NPSNET), High Level Architecture (HLA). First three were based on three basic components:

1. The object-event component. Each element in the visual simulation (virtual objects) was called 'object' or 'entity' and their interactions with other objects were called 'events'.
2. Autonomous distributed simulation nodes. Each node (participant) must place the corresponding packets onto the network for the changes-in-state of each object or entity that this participant included to the VE.
3. Dead reckoning methods. A set of prediction and convergence algorithms are used to reduce network packet traffic and synchronize the objects and their copies in other computers called 'ghosts'.

HLA is the glue that allows to integrate separate and remote computer simulations into a large simulation. Each individual simulator application is called 'federate', and a set of federates working together, through a Runtime Infraestructure (RTI) based in objects and components, are called 'federation'.

CVEs are useful Virtual Reality (VR) tools to develop Networked Robots (NR) systems where all users are running the same robotics software on their computers. In this chapter is developed a Multi-Robot-Multi-Operator CVE that needs to deal with following issues:

- GUI. Users need a set of interfaces organized in several levels of windows and menus to interact with the system.
- 3D Graphic Display. A graphics library is needed to create and display the virtual objects and the whole VE.
- Modular software. The design of an object oriented architecture, the use of a database and the implementation based in software components helps to obtain modularity. Final software must allow users to include new virtual robots, to integrate new robotics applications, and to exchange some programs for the new versions modifying as less as possible the source code.
- Distributed simulation. A middleware software must be chosen in order to get a distributed system where doesn't exist a server that performs all visual and non-visual simulation processing. The use of remote objects avoids to send long network packages reducing the low bandwidth.
- Network protocol. In a low level, middleware software must communicate between computers through a network protocol. The network protocol chosen must help to diminish the low bandwidth.
- Delays. A dead-reckoning method helps to deal with unpredictable time delays synchronizing the visual simulation on all the participants in the CVE.
- Real robot controller. Virtual robots must have a way to replicate their movements to the correponding real robots.

3 Related Work

As described in Chapter *Trends in Software Environments for Networked Robotics*, in last years NR systems in experimental robotics research have in-

creased. Robot teleoperation systems and visual simulation systems for other robotics applications are two types of NR systems. Most of those NR systems are not CVE systems, it means that they don't have a n-to-m interaction. Common teleoperation systems are networked robotics systems that have four types of interactions (defined in [GSL03]):

1. Single-operator single-robot (SOSR) teleoperation systems. A human user teleoperates the robot one at a time, it meaning a one-to-one interaction. This is the most common type of teleoperation systems. Most SOSR systems don't have a VE, for instance in Telegarden users control one manipulator [Gol00], in KheponTheWeb users work with one mobile robot [MSM97], in [DQBM98] vision tasks and teleoperation of one mobile robot are performed, and many others SOSR systems are mentioned in [GS02]. Only some SOSR systems include a VE like RoboSiM [SK99], UJI [MSDP02] and the work in [TCB04] where a virtual industrial manipulator is used to teleoperate the real robot, but where that VE is not shared with other users.
2. Multiple-operator single-robot (MOSR) teleoperation systems. Several users send different commands or positions to the same robot in a n-to-one interaction (e.g. [GSL03]).
3. Single-operator multiple-robots (SOMR) teleoperation systems. One user controls multiple robots existing a one-to-m interaction (e.g. [LHHY03]).
4. Multiple-operator multiple-robots (MOMR) teleoperation systems. Each operator controls only one robot and the robots have overlapping workspaces. This is a n-to-m interaction but where $n = m$, it means that exist n one-to-one interactions (e.g. [OKC99]).

Therefore, a CVE system that allows robot teleoperation is a n-operators m-robots (nOmR) teleoperation system for any number of operators n and robots m. In fact, the four types of teleoperation systems mentioned are particular cases of a nOmR system depending on the values of n and m. Common teleoperation systems have a client-server scheme. In a CVE is less expensive in processing and bandwidth to have a distributed scheme. This nOmR system applies for all the robotics applications that can be integrated in the CVE (e.g. teleoperation, planning, off-line programming, mapping, etc).

On the other hand, visual simulation systems including VEs are mainly single user stand-alone robotics systems used for specific robotics applications. Some of them are commercial systems like IGRIP from Delmia [Don98] that include several models of manipulators and are useful for programming and manufacturing tasks on industrial workcells, Webots from Cyberbotics [Mic04] that allows modelling and programming several wheeled mobile robots and several legged robots, etc. Other stand-alone systems that are non commercial systems are Operabotics [Say98] that is useful for path planning and teleoperation of up to four wheeled mobile robots and one aquatic robot, and Move3D [SLL01] used in motion planning for manipulators, wheeled mobile robots and legged robots. Most of those systems are used only for one user

at the same time, they are for a few number of robotics applications, and allow to work with only one or a few number of robots commonly of the same type (for example only manipulators); two exceptions of this are Move3D and Webots.

All related work mentioned above are not CVEs. We have found only a few CVEs for robotics developed mainly by two NASA laboratories:

- Virtual Environment Teleoperations Interface for Planetary Exploration (VEVI) from NASA Ames Research [HHF95] was the first CVE developed for robotics. VEVI was a software that allows to several users to teleoperate mobile robots for spatial and submarine exploration. This was a distributed system processing different modules on several computers, it means that there were multiple severs, one for the database, one for the render of the VE, one for the kinematics calculus, etc.; however, the visual simulation was not yet distributed. VEVI used the Task Control Architecture (TCA) to communicate tasks on different computers by sockets, and the commercial graphics library called World Toolkit (WTK) from Sense8 for the 3D graphic display of the VE. This system was developed only for SGI workstations and displayed until two images monitoring the remote robot actions.
- Web Interface for Telescience (WITS) from Jet Propulsion Laboratory (JPL) [BTT97] [BTN00] was used in the Mars Polar Lander Mission. WITS was a client-server architecture that allows to several researchers on different countries to work with rovers for Mars exploration. Each client computer had three module applications: teleoperation, path planning and mapping (using an stereo system). Database with all information for the simulation was in one server. WITS used remote objects to communicate with the server, Java for portability of the system on different platforms, graphics library Java3D for the 3D graphic display of the VE with VRML models, and multiple cameras for monitoring the rover.
- Mission Simulation Facility (MSF) [PLC04] from NASA Ames Research offers a simulated testing environment including robotic vehicles with manipulators onboard, terrains, sensors and vehicle subsystems. MSF uses the High Level Architecture (HLA) and is developed for autonomous tasks of rovers, spacecrafts and submarine robots. This CVE system is developed by several components that allows to generate virtual terrain surface, virtual environmental conditions, virtual robots, simulated equipment and graphical display. MSF have a database where are the models for the simulation and the results.
- Science Activity Planner (SAP) [NPV05] from JPL plans daily activities of the Spirit and Opportunity robots in the Mars Exploration Rover project. SAP is also useful as the operation interface for research rovers in develpment at the JPL. SAP renders a collection of images obtained by robots. With these images are generated terrain meshes in a 3D VE. Everybody sees the robot's environment but each user works with a dif-

ferent instance of the software to generate different plans in parallel. New information about the terrain and defined targets are updated in all the instances. SAP is implemented with Java, VRML, MySQL and XML.

The Development of a large CVE for multiple robots and multiple robotics applications, integrating the best features of all related work mentioned, involves the combination of computer graphics techniques, object oriented and components modeling, computer network protocols, distributed software, databases issues, GUIs, multithreading execution, and a lot of different algorithms for several applications. In next sections, the architecture, functionality, software implementation and examples cases of a Multi-Robot-Multi-Operator CVE are explained in detail.

4 Object Oriented Distributed Virtual Reality Architecture

Object Oriented Distributed Virtual Reality Architecture (OODVR#) is an object-oriented architecture designed for running VR applications in a distributed visual simulation system [AMMGR03] [MGAMR03].

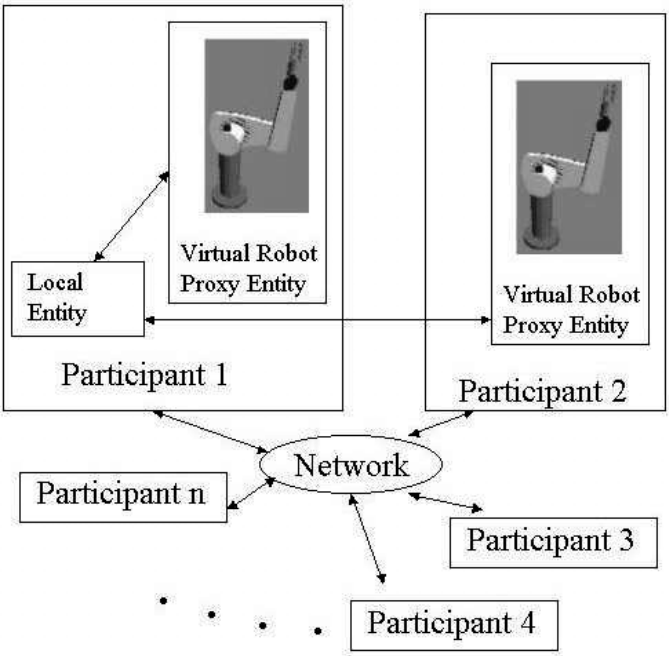

Fig. 1. OODVR# Architecture

OODVR# is composed by 'participants' and 'entities'. A participant is a new instance of the software running on a computer connected to the network, commonly each participant represents one user running the software on a computer. However, depending on computational resources, this architecture allows to one user execute one or more participants on a single computer. Entities are all objects that lie inside of the VE. Entities can be inserted in the VE at any time. The architecture is designed to have multiple entities in a distributed VE where the complete visual simulation can be accomplished by multiple participants running in several machines (see Fig. 1). This means that there is no need of a central server for the visual simulation, the entire simulation doesn't go down if one participant has some trouble. A new participant can enter to the system, at any moment during the simulation, seeing the current state of the VE. Each participant can insert zero, one or more entities in the VE. It doesn't matter how many entities a participant inserted (even zero), it can either start to work alone, to collaborate with other participants in a given task, or only see what is happening during all simulation. Entities are classified in 'dynamic entities' and 'static entities'. Dynamic entities represent objects in the VE that are able to perform movements during the simulation, for example robots and conveyors. Static entities are motionless things like floor and furniture in general.

For each entity there is a 'local entity' and several 'remote entities', called 'proxies'. A proxy entity is in charge of the 3D graphic display (render) of the virtual object that represents that entity (as in the Fig. 1 where a virtual manipulator is displayed by all the corresponding proxies). The local entity is only in the participant that inserts the entity in the simulation. For each participant there is one proxy entity, even in the participant that has the local entity. The participant who inserted a new entity is in charge of the movements (change-in-state) of that entity in all the participants through its local entity (similar to the autonomous distributed simulation nodes mentioned in section 2); it means that, if this participant starts a movement of that entity, its local entity will activate all its proxies to replicate the movement automatically in all the participants. If a participant having only a proxy entity wants to start a movement, then the proxy communicates the movement to its local entity (on the other participant), activating all proxies to execute the same movement. If for some reason, one participant that inserted entities leaves the simulation, then the next participant that moves those entities will have new local entities to control them.

This mechanism is a remote objects scheme that allows all participants in the simulation to observe multiple movements of multiple entities at the same time, doesn't matter which participant moves which entity. With this, an object oriented and distributed visual simulation is achieved, having an architecture useful to implement a CVE system.

5 System Modules

The Multi-Robot-Multi-Operator CVE System implements the OODVR# architecture adding all extra functionality needed to integrate several robotics applications. As in Brugali and Fayad [BF02], a recommendation for distributed computing in robotics is to use an object-oriented software development and a middleware framework for distributed issues. For this CVE implementation, the Java programming language is choosen in order to have a portable and multiplatform system, the Java Remote Method Invocation (RMI) is selected to manage the remote objects scheme explained in the OODVR#, and software components composed by several Java Beans are used to create a modular system. Therefore, CVE system is divided in software modules. All modules that a participant has and their main relations are shown on Fig. 2 [AMMGR03] [MGAMR03] [MGAMR04]. Main modules representing the core architecture and the basic robotics application (teleoperation) are shown inside and to the left of the big box, and will be explained in next subsections. Extra modules to the right of the big box represent other robotics applications that can use the main modules to work and will be described in next section.

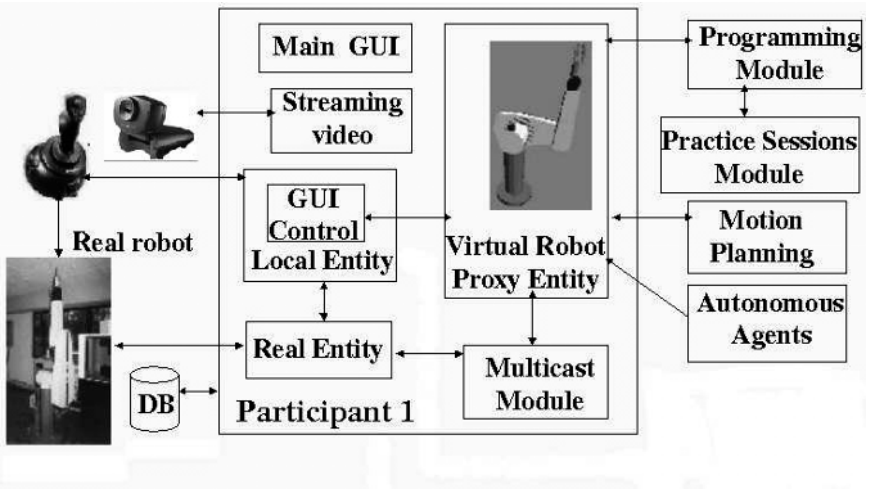

Fig. 2. CVE Modules

5.1 Core Module

This module, represented on Fig. 2 by boxes called 'Participant' (big box), 'Local Entity' and 'Proxy Entity', implements the main issues of the OODVR#

architecture (Java package called 'oodvr'): functionality of the participants, local entities and proxies as is described in section 4.

This was implemented using RMIs [PK01] but in a different way from what is common practice. It means that there isn't a common stub/skeleton approach with a client object and a server object [BF02]. Difference is that a direct communication between the stub and the remote object is performed through a Java socket (in this case, a multicast socket explained in next subsection). Is like there are no server objects, only multiple client objects controlled by the local entity (also there is no need to run the 'rmiregistry' command). Even the class *Participant* extends to the class *Entity*. Each participant has a hash table of all the other participants and other hash table for its local entities. Local entities have a hash table including all their corresponding proxies. Movements of each entity are executed in a different thread.

5.2 Communication Module

This module, represented on Fig. 2 by a box called 'Multicast Module', allows the participants to send and receive the remote objects information using a transport layer networking protocol called 'Internet Protocol Multicasting (IP Multicasting)'. For this, a Java Socket class called *MulticastSocket* is used (see [HSH99]).

IP Multicasting is a faster and more efficient protocol than Transmission Control Protocol/IP (TCP/IP), User Datagram Protocol/IP (UDP/IP) and IP Broadcasting for large-scale distributed CVEs [SZ99] [Die01]. Here are the main reasons:

- TCP/IP is the only reliable protocol of that four, but is the slownest because its packets have the biggest size and those packets may be delayed to ensure an ordering.
- UDP/IP is a not reliable protocol, therefore is faster than TCP/IP because use packets of small size and doesn't need an ordering of packets, however lost and corrupted packets happens only if the network is saturated with a lot of traffic.
- Nevertheless, both TCP/IP and UDP/IP are no so efficient during programming a final application because they are point-to-point protocols; therefore in a communication component would be necessary two things: to have always actualized a table with IP addresses of all possible computers that will connect to the CVE at any moment (this is no always possible because the locations of the computers, or will be very tedious to all users to recollect all IP addresses and give them to the software), and is needed a high network bandwith to send the same packet N times for the N computers.
- IP Broadcasting is based on UDP/IP but is a multi-point protocol that delivers the packets to all computers on a sub-network. Therefore, there is no need to have a table of IP addresses, but IP Broadcasting only can

be used in a LAN environment and still there is expensive in network bandwidth because each computer must receive and process the packet sent, even if the CVE system is not running on that computer.

- IP Multicasting is also a multi-point protocol based on UDP/IP, but solves the problems of IP Broadcasting because only sends one packet that is received only by the computers that need that packet and are joined to the multicast group (computers that are running the CVE system), preventing bottlenecks by load balancing. Information only travels through routers along networks that lead toward an interested destination host.
- Due to the scalability of a Multi-Robot-Multi-Operator CVE is better to use a faster and efficient protocol to diminish delays on the visual simulation.

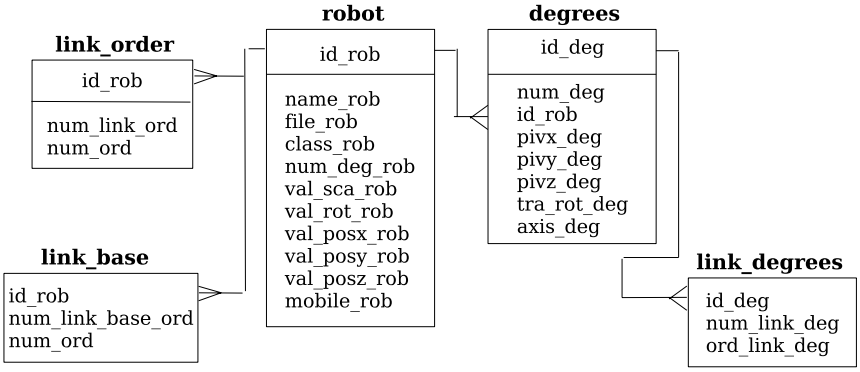

Fig. 3. Entity-relation diagram of some tables on the DB

5.3 Data Module

This module, represented on Fig. 2 by a cylinder called 'DB', is a data layer in charge of store and load information from a DataBase (DB). Robot's information stored in the DB is needed to insert a virtual robot in the VE and to move its different degrees of freedom at any moment. One example of this is represented in an entity-relation diagram shown on Fig. 3.

The tables in the figure, are the ones corresponding to the information about a robot, its geometry and its degrees of freedom. The 'robot' table has the main information about each robot: name (name_rob), number of degrees of freedom (num_deg_rob), file with the geometry (file_rob), type of robot (mobile wheeled, manipulator or legged indicated in field mobile_rob), and so forth. The 'link_order' and 'link_base' tables are used for parsering the files with the geometry of the robot. Finally, the 'degrees' and 'link_degrees' tables

have the information about each degree of freedom, the pivot point (pivx_deg, pivy_deg, pivz_deg), the type of joint (translation or rotation indicated in the field tra_rot_deg), the part of the geometry that will be moved with the degree of freedom (table 'link_degrees') and so forth. The system allows two types of geometry files: VRML and OBJ.

The implementation of this module was performed with Java DataBase Connector (JDBC) and the freeware DB MySQL. Each time that a new virtual robot needs to be added to the system it is only necessary to fill the corresponding information in the DB (without recompile).

5.4 Real Robot Module

This module, represented on Fig. 2 by a box called 'Real Entity' and by the real robot image, allows the connection of the system to each type and specific model of real robot through its own library (e.g. Aria libraries for mobile robots Amigobot and Pioneer3DX from ActivMedia, Tekkotsu libraries for Sony Aibo legged robot, etc.). Tekkotsu is already in Java. We have made the connection between Java and the corresponding C++ libraries for some real robots tested using Java Native Interfaces (JNI) [Gor98] and threads. To achieve this connection, four classes are important, three classes implemented in Java and one in C++. For example, if is required to teleoperate two degrees of freedom of an AmigoBot (translation and rotation), therefore:

- First class called *RealRobot* (made in Java) instanciate all corresponding Java classes in charge of teleoperating or controling each real robot through a Java reflexion mechanism (use the Java classes *Class* and *Method*) taking the name of the corresponding robot class from the data module. Only is needed one class of this type for all new robots added on the system.
- Second class called *AmigoBot* (made in Java) is the class that teleoperates or controls the real robot AmigoBot, instanciates the third class called *TeleoperaAmigoBot* (made in Java) and loads the fourth class also called *TeleoperaAmigoBot* (but is made in C++) via the dynamic library called *libAmigoBot.so*. Down is an example of the code structure of this class.
- *TeleoperaAmigoBot* is the class in the Java system that corresponds to the class made in C++. There must be defined as native the method that shares this class and the class in C++ (e.g. private native void IsRobot();).
- *TeleoperaAmigoBot* is the class in C++ that includes the Aria libraries functions to move the real robot. This program needs to include a '.h' file generated by the command javah (e.g. #include "TeleoperaAmigoBot.h"), and the needed methods must have specific names (e.g. JNIEXPORT void JNICALL Java_TeleoperaAmigoBot_IsRobot(JNIEnv* env, jobject thisObj)).

```
... // Here goes some packages needed
public class AmigoBot
{
  ... // here goes some variables and objects definitions needed
  TeleoperaAmigoBot teleop;
  static
  {
   System.loadLibrary("AmigoBot"); // To load the dynamic library
  }
  public AmigoBot()
  {
  }
  public void open() // Method that opens the connection to the real one
  { ...
   teleop.start();
  ... }
  public boolean isRobot()   { ...
   teleop.isRobot(); // Method that check if the real robot is available
  ... }
  public void rotate(int grades) // Sends rotate command to the robot
  { ...
   teleop.rotate();
  ...}
  public void translate(int mm) // Sends translate some distance to the robot
  { ...
   teleop.translate();
  ...}
  }
```

Code structure of class *AmigoBot*

5.5 Virtual Environment Module

VE module, represented on Fig. 2 by a virtual robot inside of the box 'Proxy Entity', is the one that performs the render of the 3D virtual objects in the VE using the Java3D libraries [BP99]. Due the slowness of Java3D when there are a lot of 3D virtual objects, some tests have been done using OpenGL libraries [SWND05] connect them to the system also with JNIs. This module allows each participant to see the same VE from the desired point of view. Left side on Fig. 4 shows an example of a VE including a workcell with four manipulators, furniture, floor and a conveyor.

5.6 User Interface Module

This module, represented on Fig. 2 by the boxes called 'Main GUI' and 'GUI Control', and the image of a joystick, implements the GUIs and the external device interfaces. GUIs display all menus, windows and options that needs

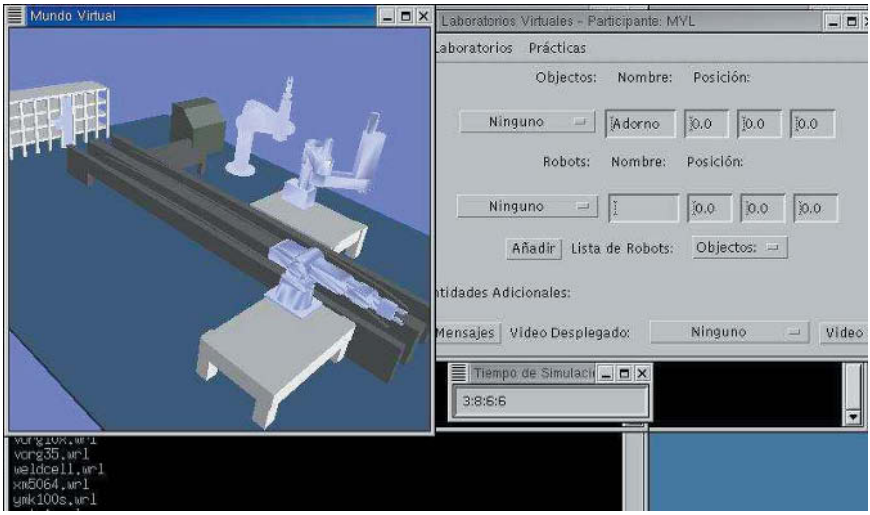

Fig. 4. VE and GUI

each user to control the virtual or real robots (right side of Fig. 4 shows the main GUI in the big window and the simulation clock in the small one). External devices include joysticks or steering wheels. This module was implemented using Java Abstract Window Toolkit (AWT) and threads.

5.7 Telemonitoring Module

This module, represented on Fig. 2 by the box called 'Streaming video' and the image of a webcam, uses videostreaming through a multicast protocol to allow remote monitoring for the real robots. This was implemented with Java Media Framework (JMF) [GT99]. Telemonitoring can be accomplished from multiple video cameras that could be connected to different participants; one camera maybe seeing the robot environment and other mounted on robot. Each participant can decide how many cameras' view will open to telemonitor the scene (even can decide only to see the virtual robot without see any camera's view).

6 Robotics Applications

Until now, we have integrated four robotics applications: teleoperation, programming practice sessions, reactive agents, and path planning. The first one, teleoperation, is achieved by the core module and the real robot module. Last three were easily implemented in our modular CVE because they use mainly previously developed modules. We need to take care of the own features of

each application, because the general functionality is done on the system (virtual robots display, control of robot movements, etc); only is needed to add some new menus and bottons to the GUI, add some tables to the DB, and add some classes to the system (as in the case of the real robot module) to connect each application.

6.1 Teleoperation

In our CVE, each participant can teleoperate multiple manipulators, mobile and legged robots (virtual or real). With the core module implementing the main issues of the OODVR# architecture, we have a natural way to perform the 'teleoperation of virtual robots'. Teleoperation of real robots is done with the real robot module. Tests were done on Linux and Windows.

A simple control mechanism can be activated in order to allow only one user to control the one real robot at the same time. In the GUI each user can take the control of that robot and none other user can move it. The user can release also the control of a robot and so take it other users.

A simulation clock allows the movement synchronization of the robots, considering that can exist delay in the network. When a user moves a degree of freedom of a robot, this information is sent to the local entity with the start time of the movement, and the speed. When a proxy receives the information about the movement, if the start time is different of the current time, this proxy will move the robot with a bigger value of speed. With this mechanism all proxies finish the movement at the same time.

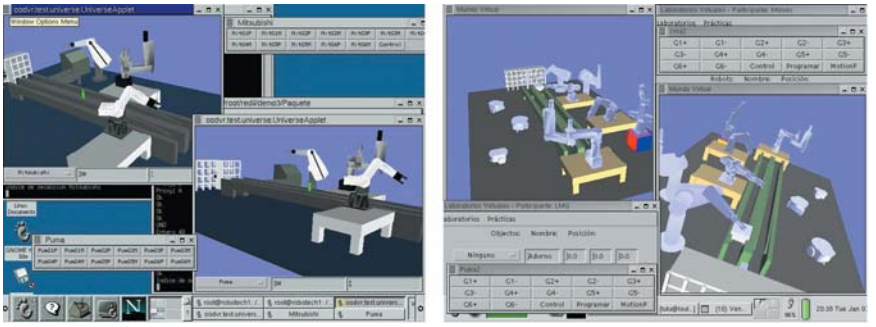

Fig. 5. Two examples of teleoperation of multiple virtual robots

Fig. 5 shows two examples of teleoperation of multiple virtual robots, both examples show two participants on the same computer (but this two participants were interacting with more participants in other machines) and the same workcell with multiple robots but each static and dynamic entity has different colors. Left side only has four manipulators (Mitsubishi Movemaster

EXR, Amatrol Jupitel XL, Amatrol ASRS and Puma560) but right side has seven manipulators (same first three manipulators than in the first example plus two Puma 560 and two CRSA 465) and four mobile robots (two Pioneers 3DX, one AmigoBot and the conveyor that works as a mobile robot). In that figure can be seen the different points of view of the VE on each participant and the GUI windows that are used to control each degree of freedom of each robot.

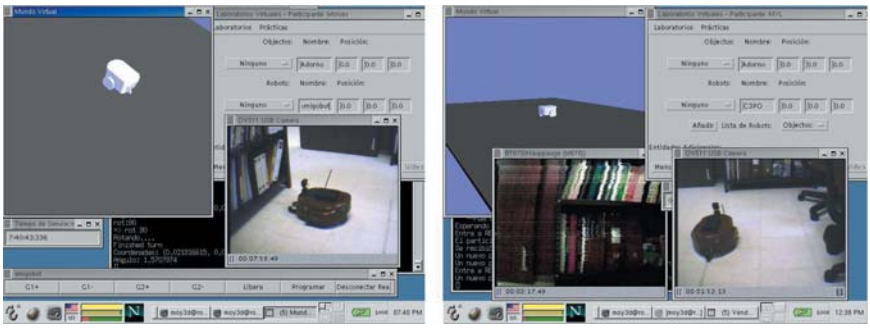

Fig. 6. Two snapshots of two different computers executing teleoperation of a real mobile robot

Teleoperation of a real mobile robot AmigoBot is shown on Fig. 6. In that figure there are two snapshots of two participants in a test with more participants. The real robot can be observed by the video sent by two cameras and there also exists a virtual representation of the robot in the VE. The two cameras are a webcam watching the environment where the robot moves and the camera onboard of the robot. While a participant moves the AmigoBot, the virtual robot will follow the same movements. Left snapshot shows the computer of the participant that is controlling the movements of the robot, this participant is only telemonitoring the robot with the webcam view. Right snapshot shows other participant monitoring the robot with both cameras.

Teleoperation of a real Aibo legged robot is shown on Fig. 7, there appears the VE window with the virtual legged robot and one image taken from the monitoring window.

6.2 Programming Practice Sessions

The idea of having practice sessions as an application of the system, is to provide training and learning tools for students or researchers improving distance learning programs, reducing investment and providing a flexible experimental framework. With this practice sessions the risk of damage caused by accidents can be reduced.

Fig. 7. Teleoperation of a real Sony Aibo legged robot

With this application, an instructor is able to define a practice session, and students (or simply other users) are able to perform programs and tasks to accomplish a practice session. This application can be considered as a Virtual Laboratory in which users have access to virtual equipment, in this particular case there are virtual manipulators and virtual mobile robots.

For the implementation of this virtual laboratory, the following features are considered:

- The DB is used for saving information about the practice sessions, for example robot positions, environments, programs and instruction files.
- The chat included in the system provides a communication channel between users in a practice session.

A programming language that can be used for both manipulators and mobile robots has been developed. There is also an interface for teaching positions to manipulators, as is performed with real robots using a teach pendant. With these tools users can write, compile and run programs for each robot and after that there is another interface that allows to coordinate several programs for different robots to simulate a more complex task [MGAMR03].

Defining and Executing Practice Sessions

In order to define a practice session, users must first create and save a VE in which the practice will take place. The VE can include furniture and different robots. After that a new practice has to be created including the name of the practice, the name of the VE and the file with the instructions. Once the practice has been defined, users must complete the assigment.

Each user connected to the simulation can program a different robot and test the program. When all needed programs are ready, only one user must define a task to coordinate all programs. From this point of view a task is considered as a set of coordinated programs to perform a more sofisticated

activity with the cooperation of several robots. The programs and tasks can be saved on a DB for future practice sessions.

Table 1. Control structures and instructions.

Control structure or instruction	**Description**
$if...else$	Decision control structure.
$while$	Cycling control structure.
$move(p)$	Moves robot to position p.
$move(f, g)$	Moves dof f by g units.
$speed(v)$	Defines speed v.
$flag(v)$	Assigns value v to the robot's flag.
$wcellflag(n, v)$	Writes value v to the nth flag of the environment.
$rcellflag(n, v)$	Reads the nth flag of the environment and stores the value in v.
$wait(m)$	Waits m miliseconds.

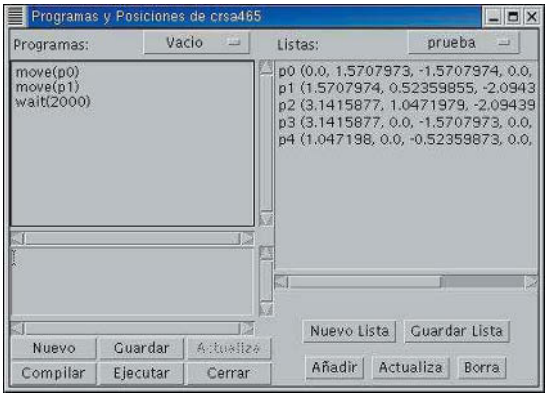

Fig. 8. GUI for programming and defining positions

For programming a robot, the GUI shown in the Fig. 8 must be used. There are two steps for programing a robot: teaching positions and writing the program. The first step, teaching positions, applies only for manipulators and is an optional step.

1. Teaching positions.
   In real manipulators, a teach pendant is used to define fixed positions that will be needed for programming a sequence of movements to that fixed positions. This feature is also implemented in the practice sessions application. The user can move each degree of freedom in a manipulator

until the desired position is reached; then the user must save the position. Each position is labeled as p_n, where n is a natural number. Once several positions are defined, the user can save a list. The list will be accesible to all practice sessions.

2. Writing a program.
   The developed programming language includes the instructions and control structures shown on Table 1. Users can create new programs and change existing ones at any time during the simulation. Each single program is saved on a file and compiled "on the fly" when the user runs the program for the first time. Each robot has an internal state flag that can be read by other robots (see instruction *flag* on Table 1). There are also a set of ten flags that corresponds to the environment that can be set and read by all robots. All these flags are useful to coordinate running programs in different robots (e.g. a robot can wait until a flag is set by other robot to continue the execution of the program). Each program is saved in a file and in the DB, and will be accesible to all practice sessions.

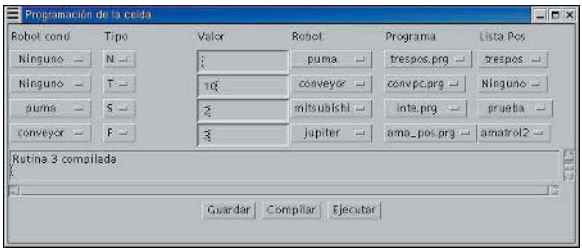

Fig. 9. GUI for defining a task with all the starting conditions

For coordinating the programs in a single task, only one user must define the starting conditions for each program using the GUI showed in Fig. 9. The interface shows six columns. The first three columns define the starting condition, and the last three define the robot, the program and the list of positions. There are four starting conditions:

- None (N): There is no condition to run the program. If all programs has no starting conditions, all programs will be executed in parallel.
- Time (T): The program will start after t seconds from the begining of the task. Time t is specified in the textbox next to the T.
- Status (S): The program will start when the robot indicated in the first column is in a given state. There are two possible states: a value of 1 means that a robot has started a program, and value of 2 means that a robot has finished a program.
- Flag (F): The program will start when the robot indicated in the first column has its flag in a certain value.

Once a user has defined a task, this can be saved for future changes. Only one task can be saved for each practice session.

Hints on Implementation

In order to develop the application previously described, several classes must be implemented. Basically these classes were divided in different components (each one is a Java package) depending on the functionality: classes for the GUI, classes for the DB connection, classes for open and saving practice session, classes for opening and saving lists of positions classes for opening, saving, compiling and running programs, and classes for opening and saving a task.

The main issue in the practice sessions application was to define a way to allow programs to be compiled and executed "on the fly", this means without having a predefined set of programs or instructions. The compiler for the programming language was developed using the Java Compiler Compiler (javacc), which is a tool that allows to define the lexical and grammatical rules using an script file and after that generates the java code for the compiler. The programming language, besides of the instructions and control structures shown on Table 1, allows to define integer variable and to perform arithmetical operations with those variables. The input of this compiler is a program of the defined programming language and the output is the java code for that program.

Inside the application, a user writes a program in the GUI and compiles it, therfore, the text of the program is saved in a file '.prg' (compiler input), the compiler is invoked and a file '.java' is written (compiler output). All programs written by users will have a *run* method in the '.java' file. After that, if the user wants to run the program in the simulation, then happens the following:

- A '.class' file is generated using the *Runtime* Java class (specifically the *exec* method). The input will be the '.java' file.
- The corresponding class is executed in the simulation with a reflexion mechanism. The '.class' file corresponding to the user's program is instanciated with the Java class called *Class*. After that the *run* method is invoked.

In this way, this mechanism is like adding new code to the simulation to perform new tasks with the robots in the VE.

Example

There is a manufacturing cell in the simulation with several manipulators (Mitsubishi EXR, Amatrol Jupiter XL and a Puma 560). The task is the following: make Puma to take a piece from the conveyor, put the piece on a lathe, and after a defined time return the piece on the conveyor again. Then move the conveyor until the piece is close enough of Jupiter robot. Jupiter

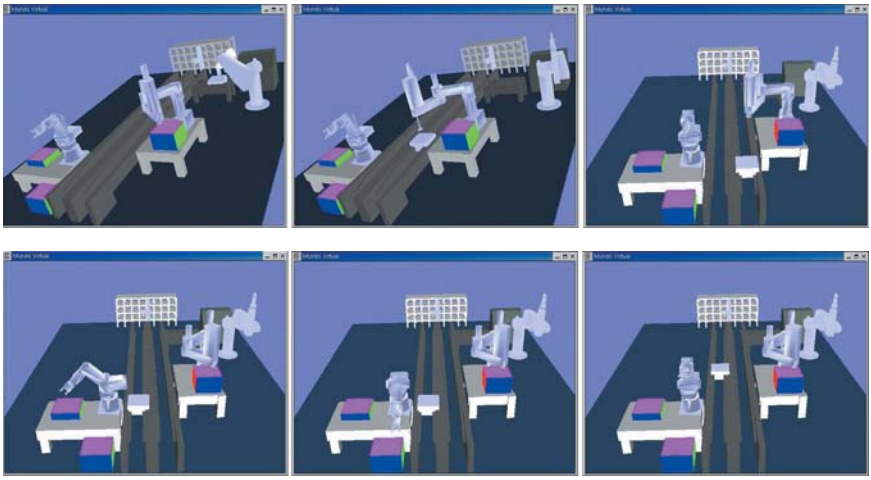

Fig. 10. Sequence of a practice session in a workcell with manipulators

robot must take a stamp from a box and put it on the piece. After that the conveyor must arrive to the Mitsubishi robot. Mitsubishi robot must take a cloth from a box and clean the piece. Finally the conveyor must return to the end. In Fig. 10 there is a sequence of snapshots taken during the execution of the task previously defined.

Down is an example of the code in the programming language defined, that is a part of the code for the conveyor in this example of practice.

```
int a  //declaration of variable a
a = 0
flag(1) //The internal state flag for the conveyor is set to 1
        //this flag will be read by other robot.
rcelflag(1,a) //Read the environment flag number 1 using variable a
//The conveyor will wait until flag number 1 is set by other robot
while(a == 0)
{
  wait(500)
  rcelflag(1,a)
}
move(p1) //move to position defined by p1
move(1,1000) //move degree of freedom 1 in 1000 units
```

Part of the conveyor code

6.3 Reactive Agents

This application was developed to add agents with certain behavior. In this particular case robots become robotics agents using a reactive architecture. The problem described by [Ste99] was selected and implemented for the reactive agents application. The problem is to explore an unknown terrain using several wheeled mobile robots and to pick up rock samples that have to be returned to a space base.

At the beginning of simulation, all robots leave the space base and start the exploration. Robots do not have a map of the terrain and do not know where the rocks are. However robots have sensors to determine if in front of it there is an obstacle or a pile of rocks, and a sensor to "hear" a radio signal emitted by the space base. When a robot finds a rock, it has to return to the space base (following the radio signal) to leave the rock and then returns to explore again. There is no communication between robots and each robot works independently, but when a robot finds a pile of rocks and returns to the space base, it leaves a trail of crumbs that can be used by it or other robot to find the pile of rocks again.

Reactive Architecture for the Robot Agents

There are modes to perform the exploration: individual and cooperative. In the individual mode, robots works independently without leaving the trail of crumbs, so this means that each robot will explore always as the first time robot left the space base. On the other hand, when robots explore and leave crumbs, the next time a robot senses a trail of crumbs will follow it and go directly to a pile of rocks. Both mode use a layered reactive architecture defined as follows:

1. Avoiding obstacles layer: Robot rotates 45 degrees to left or right at random if an obstacle is detected in front of it.
2. Taking rock to the space base layer: If the robot is in the space base, then drops the rock. In the individual mode, if the robot is not in the space base, using the radio signal determines in which direction will be the space base in order to go closer to it. In the cooperative mode, before going to the next position according to the radio signal, the robot drops a pair of crumbs.
3. Collecting samples layer: When a rock or pile of rocks is detected in front of a robot, the robot takes one rock sample.
4. Collecting crumbs layer: This layer is only defined in the cooperative mode. If the robot detects a pair of crumbs, the robot takes only one crumb and moves in the direction indicated by crumbs.
5. Exploring layer: If the robot does not detect anything in front of it, then the robot moves randomly to explore the terrain.

The priority of the layer goes from the first layer to the last one, so avoiding obstacles layer is the one with the highest priority. Because of this layers definition, agents cannot determine when there are no more rocks, so they will continue the exploration until the user stops the simulation.

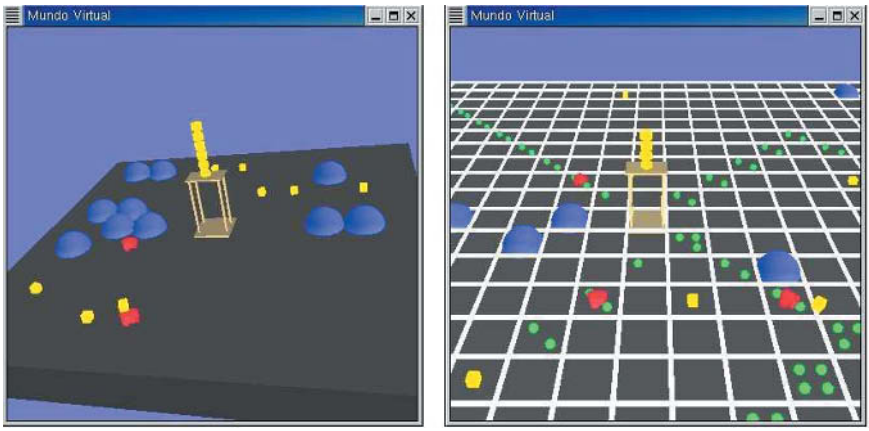

Fig. 11. Reactive Agents examples for individual and cooperative modes

In Fig. 11. there are two snapshots of the simulation, the first one is for the individual mode and the second one is for the cooperative mode. In the VE, the obstacles are represented by blue spheres, rocks by yellow cubes and crumbs by small green spheres. The space base is located in the center of the terrain and the robots are colored in red (they represent an AmigoBot).

Implementation Issues

Since this application is based on the OODVR# architecture, all objects and reactive agents that will be part of the VE are entities. The robots, the rocks and the crumbs will be dynamic entities; and the terrain, the space base and the obstacles will be static entities. In fact the rocks, crumbs and robots are considered as mobile robots in the simulation because they need to have motion capabilities in the environment.

For the integration of this application, an autonomous agent module was developed. This module models the terrain as a regular grid with cells, in which each cell contains information about the environment. Each cell has several boolean variables indicating if there is or no an obstacle, a robot, crumbs or rocks inside the cell. Robots can only sense the cell in front of it. For the radio signal emitted from the space base, in each cell there is a value indicating the proximity between the current cell and the space base, so with this information each robot can determine which is the more convenient cell

to get closer to the space base. In this case, robots can sense the radio signal in the eight neighboring cells. This information on the grid can be read by the robots in order to determine which layer has to be activated and define which action the robot will perform. Once a layer is activated, it inhibits the execution of the ones with less prioriy.

The dynamic entities (one entity for each robot, rock and crumb) are teleoperated automatically by the autonomous agent module by only sending to the core module when each entity needs to move according to the behavior specified by the active layer. In this application, the simulation runs having hundreds of entities, and the system performs better under Linux, due to the fact that each entity has its own thread and Windows didn't work well with many threads.

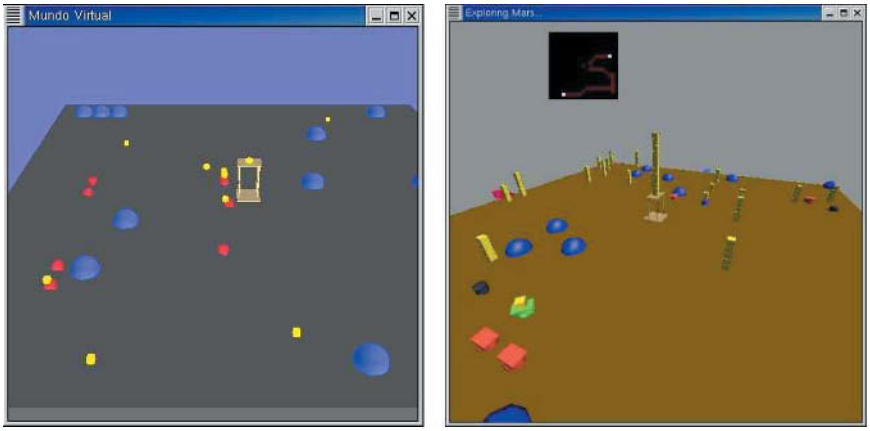

Fig. 12. Reactive Agents examples for direct communication and specialized modes

Other two different cooperative modes were implemented using direct communication between the robot agents instead of crumbs, they are called direct communication mode and specialized mode. Direct communication is performed through a blackboard in which each robot can write and read messages. In the first new mode, the only difference with respect to the crumbs case is the direct communication. In the second new mode, besides the direct communication, there are several types of specialized robots with different load capabilities and with different functions. The robots with different load capabilities can carry an specific number of rock samples more than only one. Robot with different functions are: explorer robots (black robots) and loader robots (red robots if they are not loading and green if they are). Explorer robots only find the pile of rock samples and communicate the position to the loaders robots making a map (black square) of the unknown environment. First of this last two operation modes were also implemented in Java,

but because the experiments with a lot of entities ran too slow in Java, last mode were implemented in C++ using OpenGL. Fig. 12 shows these last two modes, in the direct communication mode (left side) when one robot notify to the others one position with samples, several robots go to that position forming a line.

6.4 Path Planning

The path planning application was integrated as a new module of the system. The key idea behind integrating a path planning module is to show that some external developed systems can be integrated as module of the system with only a few modifications (component represented by the package 'motionp').

The path planning module was developed as an independent application by Arechavaleta-Servin and Swain-Oropeza in [ASSO03]. That application was easily integrated to our CVE. This application solves the problem of finding a collision free path between two positions (called initial and goal positions) of the robot in a known environment. The implemented module solves the path planning problem using three different types of roadmaps. The idea of a roadmap is to have a set of roads in the environment that allows the conection between different configurations. Usually a single roadmap is defined for an environment and when a robot wants to go from an initial position to a goal position, the robot first has to find a path between the initial position to the roadmap, then travels using the roadmap and finally has to find a path between the roadmap and the goal configuration. Here is assumed that the environment is populated with polygonal obstacles.

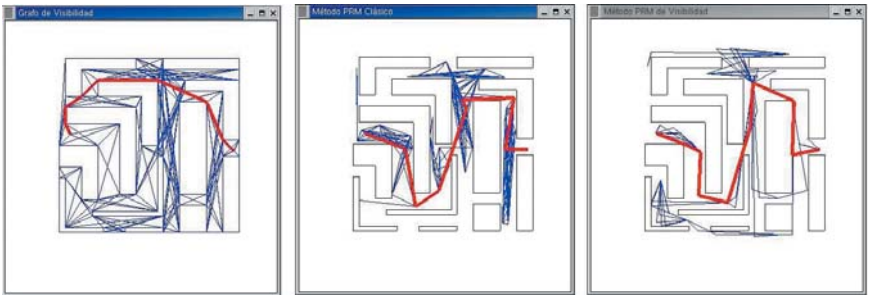

Fig. 13. Visibility graph, probabilistic and visibility probabilistic roadmaps

Three different types of roadmaps were implemented on that path planning application: Visibility Graph roadmap, Probabilistic roadmap and Visibility Probabilistic roadmap. After the roadmap is built, the Dijkstra's algorithm is used to find the shortest path in the graph. Examples of the three types of roadmaps are shown on Fig. 13.

Implementation Issues

The original path planning application was developed in C/C++ with OpenGL for the graphic display of the environment. The application was integrated as a new module, so the result is: a user with a mobile robot in the simulation will be able to see the robot following the resultant path in the VE after running the path planning module. Besides, if there is a real robot associated to a virtual one and the VE is a representation of the real environment in which the robot is, users connected to the simulation will be able to see the execution of the path in both the VE an in the real robot telemonitored.

These are classes that were modified or added in order to integrate this module as a part of the system and to allow the display of the robot performing a path in the VE:

- Add information on the data module: Information about which kind of robots will have access to this module. Until now, only wheeled mobile robots are able to use the path planning module, but this can be easily modified for manipulators and legged robots.
- Add widgets in the GUI: For the mobile robots, there is an extra button label as 'MotionP' that allows the execution of a path planning algorithm.
- Add the corresponding classes in Java: The main program of the C/C++ application must be converted into a native method using JNI, that will communicate with a Java class that is part of the virtual entity during the path following. The class *MotionPlanning* is the one that was added to the system; this class has a native method (called *RoadMap*) and when this method is invoked from the simulation the main program of the C/C++ is executed.

The main advantage of having the implementation of the path planning module in this way is that the algorithm developed in C++ can be replaced by other one without recompiling the system in Java. It is only needed to recompile the C++ code.

Fig. 14 shows three images: the real environment with the real robot, one snapshot of the simulation showing the path planning application in OpenGL

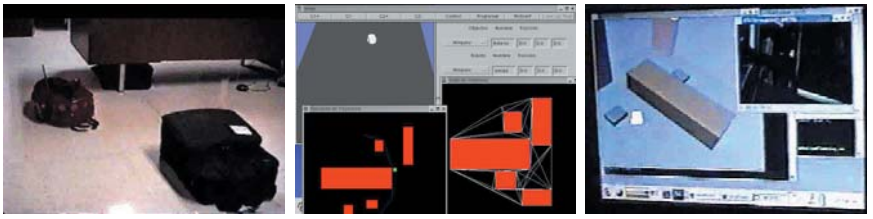

Fig. 14. Path planning for a real and virtual mobile robot

integrated in the CVE system, and the last one has the VE of the real environment and a window where the robot is monitored with its camera onboard.

7 Conclusions

We have designed and implemented a Multi-Robot-Multi-Operator Collaborative Virtual Environment useful for teaching robotics concepts and for collaborative research. In our system, we have integrated four different applications: teleoperation, programming practice sessions, reactive agents and path planning, and maybe more can be added. The main contribution is that our CVE system is the first distributed CVE for robotics applications that allows nOmR teleoperation of robots of three different types: manipulators, wheeled mobile robots and legged robots (combining SOSR, MOSR, SOMR and MOMR systems). As future work, we will perform test with students in robotics classes, to add more applications like inverse kinematics and more robots of each type defined, and to improve performance of implementation.

References

[AMMGR03] M. Alencastre-Miranda, L. Munoz-Gomez, and I. Rudomin, *Teleoperating robots in multiuser virtual environments*, Proceedings of 4th Mexican International Conference on Computer Science (September 2003), pp. 314–321.

[ASSO03] G. Arechavaleta-Servin and R. Swain-Oropeza, *Searching motion planning strategies for a mobile robot*, Proceedings of IASTED Robotics and Applications (RA'03) (June 2003), pp. 128–133.

[BF02] D. Brugali and M. E. Fayad, *Distributed computing in robotics and automation*, IEEE Transactions on Robotics and Automation Vol. 18 (August 2002), no. 4, pp. 409–420.

[BP99] K. Brown and D. Petersen (eds.), *Ready-to-run java 3d*, John Wiley & Sons Inc., New York, 1999.

[BTN00] P. G. Backes, K. S. Tso, J. Norris, G. K. Tharp, J. T. Slostad, Bonitz R. G., and K. S. Ali, *Internet-based operations for the mars polar lander mission*, Proceedings of IEEE International Conference on Robotics and Automation (ICRA'00) Vol. 2 (April 2000), pp. 2025–2032.

[BTT97] P.G. Backes, G.K. Tharp, and K.S.; Tso, *The web interface for telescience (wits)*, Proceedings of IEEE International Conference on Robotics and Automation (ICRA'97) Vol. 1 (April 1997), pp. 411–417.

[CPMTT99] T. K. Capin, I. S. Pandzic, N. Magnetat-Thalmann, and D. Thalmann (eds.), *Avatars in networked virtual environments*, Wiley, New York, 1999.

[Die01] S. Diehl (ed.), *Distributed virtual worlds. foundations and implementation techniques using vrml, java and corba*, Springer-Verlag, Berlin Heidelberg, 2001.

[Don98] D. L. Donald, *A tutorial on ergonomic and process modeling using quest and igrip*, Proceedings of the 1998 Winter Simulation Conference (December 1998), pp. 297–302.

[DQBM98] L. R. De Queiroz, M. Bergerman, R. C. Machado, S. S. Bueno, and A. Elfes, *A robotics and computer vision virtual laboratory*, Proceedings of 5th International Workshop on Advanced Motion Control (1998), pp. 694–699.

[Gol00] K. Goldberg (ed.), *The robot in the garden. telerobotics and telepistemology in the age of the internet*, MIT Press, Cambridge, Massachussets, 2000.

[Gor98] R. Gordon (ed.), *Essential jni: Java native interface*, Prentice Hall, Upper Saddle River, NJ, 1998.

[GS02] K. Goldberg and R. Seigwart (eds.), *Beyond webcams: An introduction to online robots*, MIT Press, Cambridge, 2002.

[GSL03] K. Goldberg, D. Song, and A. Levandowski, *Collaborative teleoperation using networked spatial dynamic voting*, Proceedings of the IEEE Vol. 91 (March 2003), no. 3, pp. 430–439.

[GT99] R. Gordon and S. Talley (eds.), *Essential jmf: Java media framework*, Prentice Hall, Upper Saddle River, NJ, 1999.

[HHF95] B. Hine, P. Hontalas, T. Font, L. Piguet, E. Nygren, and A. Kline, *Vevi: A virtual environment teleoperations interface for planetary exploration*, Proceedings of 25th International Conference on Environmental Systems (July 1995).

[HSH99] M. Hughes, M. Shoffner, and D. Hamner (eds.), *Java network programming: a complete guide to networking, streams, and distributed computing*, Manning, Greenwich, CT, 1999.

[KWJ99] F. Kuhl, R. Weatherly, and Dahmann J. (eds.), *Creating computer simulation systems: An introduction to the high level architecture*, Prentice Hall PTR, Upper Saddle River, NJ, 1999.

[LHHY03] Z.-J. Liu, Y.-L. Huang, P. Huang, and P. Yang, *Core data schedule in single-operator multi-agent network robot system*, Proceedings of the 2nd International Conference on Machine Learning and Cybernetics (November 2003), pp. 746–750.

[MGAMR03] L. Munoz-Gomez, M. Alencastre-Miranda, and I. Rudomin, *Defining and executing practice sessions in a robotics virtual laboratory*, Proceedings of 4th Mexican International Conference on Computer Science (September 2003), pp. 159–165.

[MGAMR04] L. Munoz-Gomez, M. Alencastre-Miranda, I. Rudomin, R. Swain-Oropeza, G. Arechavaleta, and J. Ramrez-Uresti, *Extending oodvr, a collaborative virtual robotics environment*, Proceedings of 1st Workshop on Virtual Laboratories at IX IBERO-AMERICAN WORKSHOPS ON ARTIFICIAL INTELLIGENCE(IBERAMIA'04) (November 2004), pp. 409–418.

[Mic04] O. Michel, *Webots: Professional mobile robot simulation*, International Journal of Advanced Robotic Systems Vol. 1 (March 2004), no. 1, pp. 39–42.

[MSDP02] R. Marin, P. J. Sanz, and A. P. Del Pobil, *A predictive interface based on virtual and augmented reality task specification in a web telerobotic system*, Proceedings of the IEEE International Conference on Intelligent Robots and Systems (IROS'02) Vol. 3 (September-October 2002), pp. 3005–3010.

[MSM97] O. Michel, P. Saucy, and F. Mondada, *Khepontheweb: an experimental demonstrator in telerobotics virtual reality*, Proceedings of the International Conference on Virtual Systems and MultiMedia (September 1997), pp. 90–98.

[MZP99] M. R. Macedonia, M. J. Zyda, D. R. Pratt, P. T. Barham, and S. Zeswitz, *Npsnet: A network software architecture for large scale virtual environments*, Presence Vol. 3 (Fall 1999), no. 4, pp. 265–287.

[NPV05] J. F. Norris, M. W. Powell, M. A. Vona, P. G. Backes, and J. V. Wick, *Mars exploration rover operations with the science activity planner*, Proceedings of IEEE International Conference on Robotics and Automation (ICRA'05) (April 2005), pp. 4629–4634.

[OKC99] K. Ohba, S. Kawabata, N. Y. Chong, K. Komoriya, T. Matsumaru, N. Matsuhira, K. Takase, and K. Tanie, *Remote collaboration through time delay in multiple teleoperation*, Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (October 1999), pp. 1866–1871.

[PK01] E. Pitt and McNiff K. (eds.), *java.rmi the remote method invocation guide*, Pearson Education, Harlow, England, 2001.

[PLC04] G. Pisanich, L.Plice, C.Neukom, L. Flckiger, and M. Wagner, *Mission simulation facility: Simulation support for autonomy development*, Proceedings of 42nd AIAA Aerospace Science Conference (January 2004).

[Say98] C. Sayers (ed.), *Remote control robotics*, Springer-Verlag, New York, 1998.

[SK99] A. Speck and H. Klaeren, *Robosim: Java 3d robot visualization*, Proceedings of 25th Annual Conference of the IEEE Vol. 2 (November-December 1999), pp. 821–826.

[SLL01] T. Simeon, J. P. Laumond, and F. Lamiraux, *Move3d: a generic platform for path planning*, Proceedings of 4th IEEE International Symposium on Assambly and Task Planning (May 2001), pp. 25–30.

[Ste99] L. Steels, *Cooperation between distributed agents through self organization*, Proceedings of 1st European Workshop on Modelling Autonomous Agents in a Multi-Agent World (1999), pp. 175–196.

[SWND05] D. Shreiner, M. Woo, J. Neider, and T. Davis (eds.), *Opengl programming guide*, Addison Wesley, Upper Saddle River, NJ, 2005.

[SZ99] S. Singhal and M. Zyda (eds.), *Networked virtual environments*, Addison Wesley, Reading, MA, 1999.

[TCB04] J. Tan, G. J. Clapworthy, and I. R. Belousov, *The integration of a virtual environment and 3d modelling tools in a networked robot system*, International Journal of Image and Graphics (October 2004), pp. 1–20.