Université d'Ottawa · University of Ottawa

Université d'Ottawa · University of Ottawa

FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTDOCTORALES

FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES

Stéphane DODELLER

AUTEUR DE LA THÈSE - AUTHOR OF THESIS

M. A. Sc. (Electrical Engineering)

GRADE - DEGREE

Department of Electrical Engineering

FACULTÉ, ÉCOLE, DÉPARTEMENT - FACULTY, SCHOOL, DEPARTMENT

TITRE DE LA THÈSE - TITLE OF THE THESIS

Transport Layer protocols for Haptic Virtual Environments

N.D. Georganas

DIRECTEUR DE LA THÈSE - THESIS SUPERVISOR

CO-DIRECTEUR DE LA THÈSE - THESIS CO-SUPERVISOR

EXAMINATEURS DE LA THÈSE - THESIS EXAMINERS

P. Liu

S. Shirmohammadi

J.-M. De Koninck, Ph.D

LE DOYEN DE LA FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTDOCTORALES

DEAN OF THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES

Transport Layer Protocols for Haptic Virtual Environments

by

Stéphane Dodeller

Ottawa-Carleton Institute for Electrical and Computer Engineering
School of Information Technology and Engineering
University of Ottawa

A thesis submitted to the
Faculty of Graduate and Postdoctoral Studies
in partial fulfillment of the requirements for the degree of

Master of Applied Science in
Electrical Engineering

Your file Votre référence
ISBN: 0-494-01462-8
Our file Notre référence
ISBN: 0-494-01462-8

Abstract

Collaborative Virtual Environments require the exchange of update messages between remote locations. Network impairments such as delay, delay variations and packet loss impede collaboration, especially when haptic devices are used.

This thesis compares three transport-layer protocols that address this concern, and describes the framework designed to evaluate them. Two previously proposed protocols have been studied (based on key updates acknowledgment and updatable queue on the sender side, respectively), and a new protocol is proposed.

This new protocol adds buffering on the receiver side. By reducing the delay variation, it significantly helps collaboration, whereas it has been shown that buffering on the sender side should be avoided.

Two applications that make use of a haptic device have been designed and implemented for the tests: a tracheostomy simulation and a box carrying application.

Résumé

Les Environnements Virtuels Collaboratifs supposent le passage de mises à jour entre différents sites. Les défauts du réseau comme le retard, les variations de retard et les pertes de paquets gênent la collaboration, surtout quand des outils haptiques sont utilisés.

Cette thèse compare trois protocoles de la couche transport, à l'aide d'un outil spécialement développé dans ce but. Deux protocoles préexistants ont été étudiés (fondés respectivement sur l'accusé de réception de messages particulièrement importants et sur une file d'attente modifiable pour l'émetteur), et un nouveau protocole est décrit.

Ce nouveau protocole retarde volontairement les messages reçus, afin de réduire les variations de retard, ce qui améliore la collaboration. Nous avons montré que la mise en attente par l'émetteur, au contraire, devrait être évitée.

Deux applications qui utilisent un outil haptique ont été développées: une simulation de trachéotomie et un transport collaboratif d'objets.

Acknowledgments

I would like to thank my supervisor, Dr. Georganas, for the way he has guided and supported me along those two years. Not only did he encourage me to consider the works and ideas that proved interesting and fruitful and became the basis of the work presented here, but he has also been attentive to my questions, most of the time answering them before I even had the time to ask, and also giving me feedback at an amazing speed. Working for him has been a pleasurable and very enriching experience.

Working at DISCOVER Lab would not have been the same without my fellow "discoverers", who have built a great social environment.

Mojtaba gave me invaluable advices and support (so much that he almost became a second supervisor to me), as well as many pointers and technical help. I am not sure that I will ever be able to return him all the favors he did to me. Thierry helped me a lot in the experiments, and I would not have been able to carry them out without him. François did a wonderful job as lab manager, never failing to help me with my hardware and software requirements.

I must finally acknowledge the support I received from my parents, for without them I would not have been able to come to Canada. Their moral support has been tremendously helpful in the times of doubt every graduate student encounters.

Contents

List of Figures

List of Tables

List of Abbreviations

CVE	Collaborative Virtual Environment
DISCOVER Lab	Distributed and Collaborative Virtual Environments Research Laboratory
DLL	Dynamic Loadable Library
DOF	Degree of Freedom
FIFO	First In First Out
GHOST®	General Haptic Open Software Toolkit
GPS	Global Positioning System
GUI	Graphical User Interface
HARMONIE	Haptic, Augmented Reality Multimedia for Networked Interactive Environments
HLA	High Level Architecture
JNI	Java Native Interface
LAN	Local Area Network
NCIT	National Capital Institute for Telecommunications
NTP	Network Time Protocol
OOS	Out of Synchronization time
OSI	Open System Interconnect
QoS	Quality of Service
RTCP	Real Time Control Protocol
RTP	Real-Time Transport Protocol
RTP/I	Real-Time Application Level Protocol for Distributed Interactive Media
RTT	Round Trip Time
SCTP	Synchronous Collaboration Transport Protocol
SRTP	Selectively Reliable Transport Protocol
TCP	Transmission Control Protocol
TOAST	Transport Layer Optimization for Applications in Surgery and Training
UDP	Uniform Datagram Protocol
VE	Virtual Environment
VR	Virtual Reality
VRML	Virtual Reality Markup Language
WAN	Wide Area Network

Chapter 1

Introduction

Virtual Reality (**VR**) has been of interest for electrical engineers since more than a decade ago (the first IEEE International Symposium on the topic took place in September 1993), and a specific field of study at the University of Ottawa since 2001, with the creation of the Distributed and Collaborative Virtual Environments Research Laboratory (**DISCOVER Lab**).

A computer application belongs to virtual reality if it provides the user with an interactive universe, which can be a number of things, from the cockpit of a science fiction starship to a set of atoms that the user can assemble to form molecules. Sometimes the "virtual" universe is actually the real world (a building you are entering for the first time), overlaid with relevant computer-generated data (arrows pointing to the direction of the user's next meeting). This is called *augmented reality*.

A Virtual Environment (**VE**) is the machine-readable description of such a virtual universe. This description is not static. The virtual environment evolves over time as a result of both timed events and user inputs. If the virtual environment is a room containing a box and a clock, the box will move when the user pushes it (user input), and the hands of the clock will move as time goes by (timed event).

1.1 Input and Output devices for Virtual Environments

Virtual environments usually are, or try to be, *immersive*, i.e. they make the user "step inside" the application. This trend has prompted the development of new devices that offer a more practical, more instinctive and more realistic way to exchange information with the computer than the mouse, keyboard or monitor. Such devices include:

- Head-mounted displays, in which two small high-resolution LCD monitors are placed in front of the user's eyes, and display a 3D world that fills the whole vision area;
- Polarized goggles, that create 3D graphics, usually on a wide screen or a set of screens;
- Datagloves and trackers that capture the movement of the body in the space. If trackers register the movement of the user's head, the picture shown by a head-mounted display can be updated accordingly;
- Haptic devices, which provide *force feedback.* The term "haptic" comes from the Greek word *haptikos*, which means "able to sense or touch". Virtual objects can be felt and manipulated through haptic devices.

Haptic devices rely on force feedback. Imagine holding a pen and moving it through thin air: no force is felt except the gravity. If the pen touches a table, it will not be able to go through it because of the reaction force the table exerts on the pen. No matter how hard one pushes the pen, the pen pushes the manipulating hand back. Haptic devices simulate such behavior by having the pen sending a force to the hand, as if it was touching something. They can also simulate soft structures (imagine the feeling you would get by pushing a pen against a sponge), or even different roughness of surfaces (to distinguish, for example, silk from sandpaper). Haptic devices are at the same time input devices (they can send their position and the force applied by the user to a computer) and output devices, the output being the force (and, depending on the device used, torque) generated.

1.2 Virtual Environments Applications

1.2.1 Networked Simulations

Historically, the first networked virtual environments have been distributed simulations. The US Department of Defense prompted the development of such simulations for its own needs, so that tank and helicopter operators may train themselves as teams, in a virtual environment where they can see each other's actions and interact (the main interaction being here shooting the enemy). The first simulator, SIMNET, has been developed between 1983 and 1990. It spawned several evolutions, the latest one being the High Level Architecture (**HLA**), a standard devised for cooperation of heterogeneous distributed simulations. In parallel, other standards have been developed by universities as well as in the industry.

Other applications, though less realistic, can also be classified under the terms "distributed simulation": networked games. Again, several players evolve in a virtual universe, and through the data exchanged on the network, are able to interact (again, the interaction here resides in the ability to shoot at each other).

The main shortcoming of such distributed simulations is that participants cannot *collaborate*. It is possible to shoot someone, for example by sending to another participant the message "I just shot you" (of course, this assumes that the participants do not cheat and that the machine sends such a message only if the target was actually shot), but it is not possible for two participants to shake hands, to carry a heavy object together, or to repair a broken engine. Those are not a mere upgrades one could add to the system, these features raise whole new issues.

1.2.2 Telesurgery

Robotics and haptics allowed the apparition of another very exciting application for virtual environments: telesurgery. If a surgeon is to act on a remote patient, force feedback is crucial. Having cameras that capture and transmit what the surgeon would see were he physically close to the patient, and a slave robot that copies the movements made by the surgeon on a master control device are not enough: a surgeon has to feel what he is doing, and needs to take the patient's reaction into account. Therefore, the input device manipulated by the surgeon has to be haptic, and to copy the reactions felt by the slave device that acts on the patient.

Some surgeries are easier to transpose in a virtual environment than others: the minimally invasive surgeries. In such surgeries, a natural orifice of the body or a small incision (less than 1 cm) is used to introduce several tools and a mini-camera inside the body. They go to the operation site, without exposing the latter directly through a large incision. This is done, for instance, in laparoscopy [1] and intravascular surgery [2]. Those techniques have been developed because they are much less invasive than traditional surgery. The drawback is that the surgeon does not operate its tools directly, but has to manipulate them through handles. Those handles are connected to the tubes inserted in the patient's body, themselves containing the tools. Given that the surgeon already lost "direct contact" with the patient, it is not very difficult, conceptually speaking, to have the surgeon operating such tools remotely.

The first telesurgery has been reported in 1993, and there have been a few successful surgeries from that point on, sometimes with the surgeon and patient physically separated by one ocean. Another development of telesurgery is surgery simulation and remote training. A tracheostomy training application is presented in this thesis, and surgeries have been simulated between Canada and Australia, for instance [3]. Other research efforts include the modeling of human tissues and tool actions, the tools being scalpels or surgical hooks [4].

1.2.3 Collaborative Design

Haptic devices can also be used to help designers in their job. To design the shape of a new product, for example a new toy, designers need to sculpt and decorate a molding matter, like clay. This can be done virtually, thanks to haptic devices [5]. Designers become able to show their new creation to more people more rapidly, by sending a 3D description of their idea rather than sending the actual piece of clay. The key advantage of such applications is that designers can collaborate when creating a product, even if they are physically distant. Not only can they discuss about the model they are observing, but they are also able to alter it in real time, if they want to improve it, or test other options. Some major toy and shoe companies have adopted such systems.

1.3 Networked Virtual Environments

The idea of sharing a common universe over a network came pretty quickly to VEs users and developers. A *networked virtual environment* can be defined as

> a software system in which multiple users interact with each other in real-time, even though those users may be located around the world [6].

Rather than evolving on its own in a virtual environment, why not share the description of the universe with other users?

Having a common description for the virtual universe at the beginning of the session gives the users a *shared sense of space.* Such a description can be sent as a file before the application begins. Participants should also have a *shared sense of presence.* This is usually achieved by representing each user by its "avatar", a computer-based model of himself that will follow his movements. Those avatars' movements, and the actions they take that modify the universe, should happen at the same time for everybody, to provide a *shared sense of time.* Facial expressions, voice or typed text can be carried via network protocols designed for multimedia communications, to provide a *way to communicate.* A networked virtual environment would seem eerie if a user could see everyone else, without being able to share advices, orders or instructions with them. Finally, the networked virtual environment should provide a *way to share*, so that users can interact with each other. Those five criteria qualify a networked virtual environment [6].

Collaborative Virtual Environments (**CVE**) are evolutions of the networked VEs. A virtual environment is said to be collaborative if it is networked, and if users are able to collaborate, to interact with each other in real time. Such a thing may seem obvious in the real world, but becomes an issue as soon as a computer network is involved.

1.3.1 Network Impairments

Consider a handshake. In the real world, when two persons shake their hands, each one adjust the position of his hand in real time, up to the point where the hands eventually meet. Now, let us imagine a situation where two persons, A and B, want to shake their hands, with a delay between the movement of a hand and its perception by the other person's eye. If A's hand is too high, and B's hand too low, A decides to lower his hand. B, unaware of this fact, thinks that A has not moved, and heightens its own hand. In the final situation, A's hand is too low, and B's hand too high, and the handshake is still not possible. If both A and B keep doing that, their hands keep oscillating between the high and low positions, never stopping at the place where they should meet. This example shows what kind of problems the network's impairments may cast in a seemingly trivial task.

Another problem arises if messages are lost between A and B. If A wants to shake B's hand and moves his hand to do so, and if the message giving the new position of A's hand is lost on its way between A and B, B will never be aware that A moved his hand in the first place. It should be noted that some messages are more important than others, and therefore more sensitive to loss. This will be developed in a subsequent chapter.

In this thesis, four network parameters are considered:

- Delay: this impairment exists in any computer network, since electrical and optical signals have a finite propagation speed. In haptics and collaborative virtual environments, it creates problems such as the one highlighted above;

- Jitter: this is the variation in delay. Studies have shown that a small jitter impedes collaboration as much as a big delay. Collaboration can become as difficult over a network whose delay varies between 0 and 20 milliseconds as over another network with a fixed 200 milliseconds delay;

- Packet loss: the network may be unreliable and drop packets;

- Bandwidth requirement: each update message, individually, represents only one packet (48 bytes plus the protocols overhead to transmit six double-precision floating point numbers, which makes 90 bytes if UDP/IP is used over Ethernet). This might appear as a small size. However, haptics require an update rate much bigger than, for example, video. If 90 bytes are transmitted every millisecond (i.e. at a 1kHz rate), the bandwidth requirement becomes 90 kbytes/s, which is not insignificant (half the bandwidth of a T1 line). This requirement is neither an impairment nor a problem by itself, but remains a concern nonetheless.

Among the communication protocols that have been proposed for VEs in the literature, the focus of this thesis has been lain on the synchronous collaboration transport protocol, that implements reliability for key update messages, and ensures that those key update messages are received on time. Another protocol that buffers update messages on the sender side has also been implemented, with the hope that it could address the bandwidth requirements issue.

1.4 Contributions of this thesis

This thesis presents a work that is part of the Haptic, Augmented Reality Multimedia for Networked Interactive Environments (**HARMONIE**) project, funded by the National Capital Institute for Telecommunications (**NCIT**) [7]. Its main goal is to provide a haptic development architecture, independent from the application, so that the wheel does not need to be reinvented each time a new distributed application that uses haptics is developed.

A contribution to this project has been made by evaluating and optimizing transport layer protocols that support distributed haptic applications. To do so, we have:

- Studied the CVE applications' particular needs in terms of communications;
- Designed and implemented suitable protocols;
- Implemented a testbed for evaluation;
- Implemented two applications using haptics in collaborative virtual environments.

We have found that:

- In the presence of jitter, buffering the update messages at the receiver site helps collaboration;
- Update messages aggregation at the sender side, on the other hand, is not efficient in a collaborative framework.

Based on those findings, a new protocol that extends the synchronous collaboration transport protocol by buffering the update messages once they are received, so that a constant delay is experienced, is proposed.

A conference presentation has been extracted from this work, and has been accepted for the 22nd Biennial Symposium on Communications at Queen's University, in June 2004.

1.5 Thesis outline

The remainder of the thesis is divided as follows:

- Chapter 2 presents background knowledge on CVEs, describes some CVE technologies that have been used in this work, and states the network issues that arise in such virtual environments. Current transport protocols devised for CVEs are reviewed;

- Chapter 3 presents the transport protocols that have been chosen and implemented in this work. The specification of each protocol is given, and a theoretical comparison between them is made;

- Chapters 4 and 5 deal with the implementation of the testbed and the test applications, respectively. An appendix gives more details on the implementation that should be interesting to someone who carries on this work;

- Chapter 6 is devoted to the results yielded by this implementation, and their analysis, that provides the contributions mentioned above.

- Finally, one chapter is devoted to concluding remarks.

Chapter 2

Background

2.1 The Architecture of a Virtual Environment

To display a Virtual Environment, a computer application needs:

- A description of the environment and the objects in a machine-readable format. The basic descriptions are shape of objects, their colors, lightning properties, positions and behaviors;
- A loader that reads these descriptions and adds the virtual objects to the virtual universe;
- A graphic rendering program that displays the virtual environment at a frame rate that gives the user the illusion he is moving. It must provide primitives for manipulation and representation of geometric objects in a three-dimensional universe;
- An interface with the user;
- An interface with haptic devices. Haptic devices often need a much higher update rate than the graphical rendering (1 kHz vs 10 to 100 Hz). This is why two separated scenes are usually created (a haptic scene and a graphic scene), and updated at different rates. The application has to maintain synchronization between both scenes;
- An interface with other virtual environments, or with other users sharing the same VE;

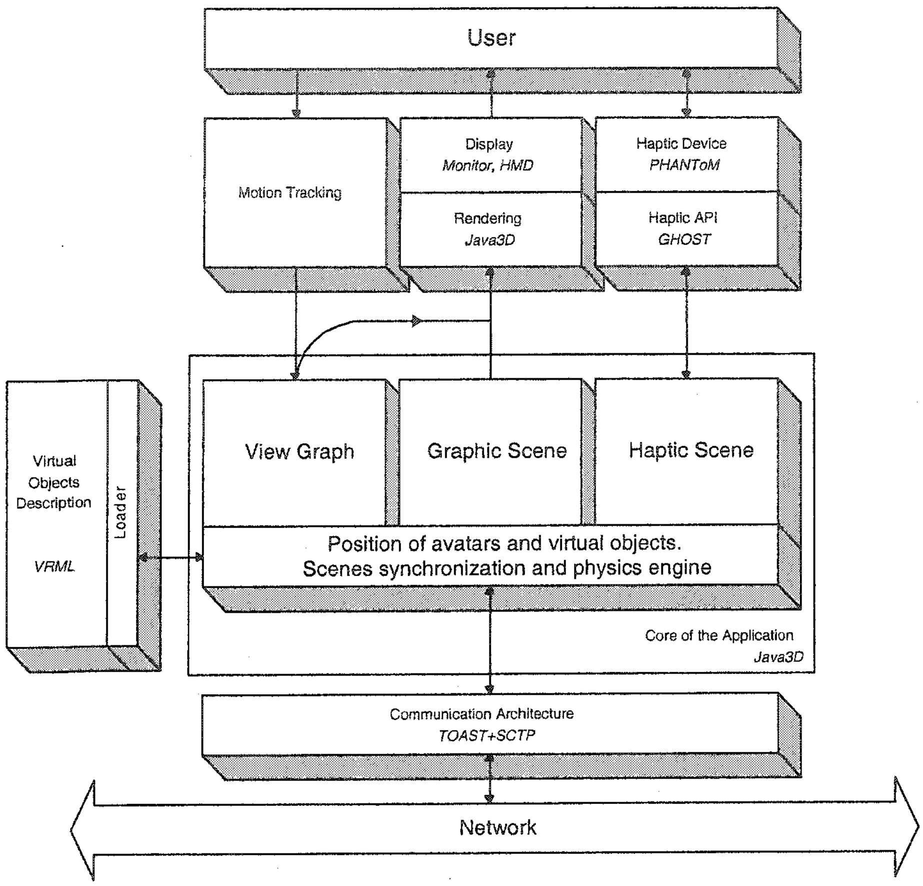

Figure 2.1: VE Application Architecture.

- Management of the interaction between the objects (collision detection) and of the physical laws (*physics engine*).

Figure 2.1 shows how those components interoperate, and gives in italics examples of tools that can be used for their implementation.

2.2 Graphic Scene

The graphic representation of the virtual world is a fundamental requirement for every VE application. Here we present the technological solutions used in this thesis to meet that requirement.

2.2.1 VRML

The graphic scene of a VE needs a way to be described and a rendering mechanism to be displayed. Descriptions of virtual objects can be either hard-coded directly in the rendering architecture (this is usually acceptable for simple shapes), or stored in an external file. One format used for those files is the Virtual Reality Modelling Language (**VRML**) [8]. This format uses a markup syntax to describe basic shapes (cube, cone, sphere and cylinder) and sets of points, their properties (color, texture), position, and lighting. Objects can be embedded into one another in a hierarchy, thus forming logical clusters of objects that will move together.

2.2.2 Java3D

Choice Justification

Java3D has been chosen to implement the rendering part of our work, for its compatibility and easy interfacing with Java, which has been used to implement the transport layer protocols and comparison framework. Java and Java3D are developed and maintained by SUN Microsystems and provide a well-documented API with numerous tools needed for such an implementation [9].

Scene Description

Java3D instantiates a universe, which contains a viewer object associated with a *canvas*. The canvas renders the 3D world in a two-dimensional Frame that can be added to a Java graphic interface, like an instance from javax.swing.JFrame for example.

Objects are then added to the universe using the following containment hierarchy, given in a top-down approach [9]:

- The universe: it contains a locale, a viewing platform (which defines the position and orientation of the viewer's eyes) and a viewer (for rendering);
- A locale: a locale is a set of high-resolution coordinates which contains Branch group-rooted subgraphs;
- Branch group: those nodes can be seen as logical units in the universe. For example, one avatar (representation of a human body in a virtual environment) could be described in a branch group. Branch groups may be

added to or removed from other branch groups, and added to or removed from the locale at runtime (live branch group). In our application, an MPEG-4-based recording and playback architecture [10] parses the VRML descriptions of the objects and adds them to the Java3D universe as new branch groups;

- Transform group: a transform group is used to group several objects that have to move together. A transform group is added as a child either to another transform group or to a branch group. One typical example is the arm of an avatar: if the whole avatar is put into a branch group, a transform group corresponding to the shoulder is added, having as children the arm and another transform group for the elbow. In turn, the elbow contains the forearm, and another transform group, the wrist, and so on. Therefore, when the angle of the shoulder changes, the positions and orientations of the arm, forearm, and hand change. If the elbow rotates, only the forearm and hand are affected. Unlike branch groups, transform groups cannot be added or removed freely. Once the branch group they belong to is added to the scene, no addition or deletion is possible;

- Primitive geometries and Shape3D: Primitive geometries are such shapes as boxes, cones, cylinders and spheres. Shape3D is the container of a geometry (usually specified as a set of points, lines, triangles or squares). Shape3D and primitive geometries are added as children of a transform group, and modified according to the transformations of their parent.

The Transformation Matrix

The transformation associated with a transform group is held in the class Transform3D. A transform3D is essentially a 4*4 matrix that can be applied to any point or vector that has to be transformed (as explained in the Java3D API) [9]:

$$\begin{pmatrix} m_{00} & m_{01} & m_{02} & m_{03} \\ m_{10} & m_{11} & m_{12} & m_{13} \\ m_{20} & m_{21} & m_{22} & m_{23} \\ m_{30} & m_{31} & m_{32} & m_{33} \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} = \begin{pmatrix} x' \\ y' \\ z' \\ w' \end{pmatrix}$$

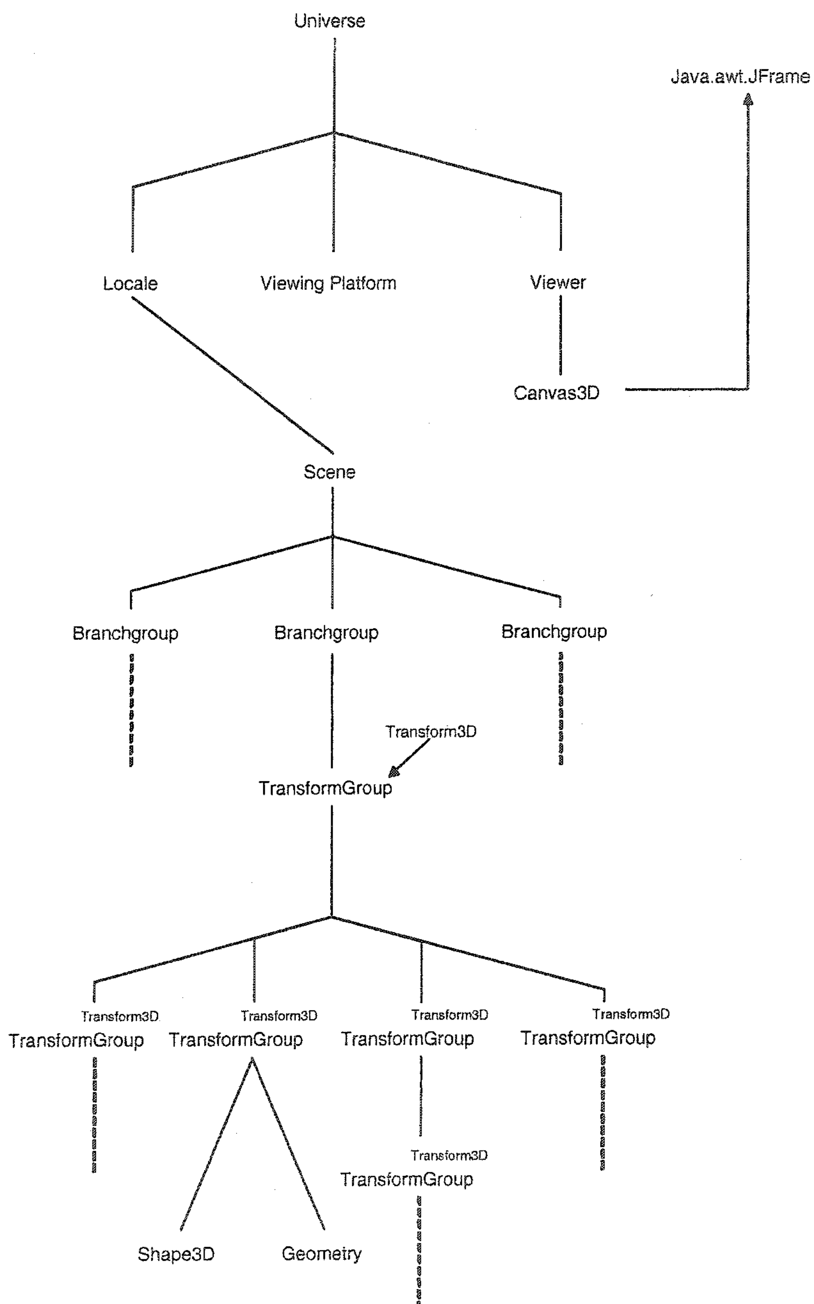

Figure 2.2: Java3D Containment Hierarchy.

with the resulting coordinates:

$$\begin{pmatrix} x' = m_{00}.x + m_{01}.y + m_{02}.z + m_{03}.w \\ y' = m_{10}.x + m_{11}.y + m_{12}.z + m_{13}.w \\ z' = m_{20}.x + m_{21}.y + m_{22}.z + m_{23}.w \\ w' = m_{30}.x + m_{31}.y + m_{32}.z + m_{33}.w \end{pmatrix}$$

Therefore, given the transformation matrix and the original point or vector, the transformed point or vector can be found. In the case of vectors, w is set to 0, and only the 9 coefficients in the upper left corner of the matrix are relevant. For example, the following matrix describes a 30° rotation around the z axis:

$$\begin{pmatrix} \cos(30^\circ) & -\sin(30^\circ) & 0 \\ \sin(30^\circ) & \cos(30^\circ) & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

To transform points, w is set to 1 and the last column becomes relevant. For example, if all the coefficients of the transformation matrix are set to zero except m_{03}, m_{13} and m_{23}, with a point to transform having the original coordinates (x, y, z), the new point is situated at $(x+m_{03}, y+m_{13}, z+m_{23})$. Therefore the transformation applied is a translation of vector (m_{03}, m_{13}, m_{23}).

In an application where all the objects are solids (keep the same scale and cannot be sheared), translations and rotations are the only transformations that can be applied.

Rendering Mechanism

In the Canvas3D that has been used in this work, rendering is entirely independent from the application programmer. However, a basic knowledge of the rendering process is useful to synchronize it with the updates of the scene description made by the application. If no synchronization is made, nothing prevents the scene to be updated several times while there is no rendering, which is useless and expensive in terms of CPU. If the demand for CPU grows and reaches 100%, the computer can no longer sustain the requirements of the application and the frame rate drops. Therefore, synchronization between rendering and scene updating can noticeably improve the graphic performances perceived by the user.

The rendering thread has the following structure [9]:

- Clear canvas;
- Call *preRender*;
- Set the view (this depends on the viewing platform position and orientation);
- Render opaque objects (each canvas pixel's color is set according to the relative position of the scene's object to the viewing point and their appearance);
- Call *renderField*;
- Render transparent objects;
- Call *postRender*;
- Synchronize the canvas with an on-screen buffer and swaps its content to this on-screen buffer;
- Call *postSwap*.

Hence, there are two buffers, and the rendering thread works on the off-screen buffer (the canvas), before swapping its content, once the picture to display is updated, to the on-screen buffer.

preRender, *renderField*, *postRender* and *postSwap* are empty methods. If Canvas3D is subclassed, those methods can be overridden to place application-defined code in the rendering thread.[1] This possibility has been used here for screen capture (copying the content of the on-screen buffer in *postSwap* and saving it as a JPEG file) and for synchronization between rendering and haptic interface polling.

2.3 Haptic devices

Haptic devices, when used over a network, highlight collaboration problems. This is why the test applications presented in this thesis include haptics in the user interface. This section presents the device that has been used and its related software.

[1] Those methods should be kept as short as possible, since the graphic window is not refreshed during their execution.

2.3.1 PHANToM

The PHANToM is one of the most common haptic devices used in VR research. Its usefulness has been demonstrated in a complex disassembly task simulation [11]. It is a pen-shaped device, with a force applied to the tip of the pen through the base of the device and the arm linked to the pen (see figure 2.3, where the arrow shows the position of the gimbal encoders).

It has six degrees of freedom as an input, and three degrees of freedom as an output. The notion of Degree of Freedom (**DOF**) can be described as the number of *independent* variables needed to describe the state of a physical object. For example, consider a solid object, too heavy to be lifted, laying on the ground. It can be moved back and forth, as well as sideways, but is constrained to stay on the ground. Therefore it has two DOF in translation. It can also be rotated around the vertical axis, but not tilted. It has therefore one DOF in rotation. In total it has three DOF: three variables exactly are needed to describe its position. A solid object has at most 6 DOF that describe its position and orientation. An articulated object may have much more than 6 DOF (for example, in a hand each finger joint brings one or two DOF).

The PHANToM can sense the movements of the user hands in all the directions of space, as well as the rotations the hand makes. It has, therefore, 6 DOF in input (three for the translation, three for the rotation).

For the output, however (the force feedback part), only the translation components may be specified. It is possible to constrain the position of the pen-part of the device in all directions of space, but rotations cannot be constrained. It is possible to make the user feel a force, but not a torque. It should be noted that other haptic devices provide six DOF both in input and output, but they are considerably bulkier and more expensive (See, for example, the MPB 6DOF used in remote surgery simulation [3]).

2.3.2 Architecture of a Haptic Device Interface

The General Haptic Open Software Toolkit (**GHOST®**) API has been developed by the manufacturer of the PHANToM to provide a C++ interface with the hardware [12]. Our purpose here is not to make an extensive description of this API, but to highlight the few features we used in our implementation, along with the code that has been used.

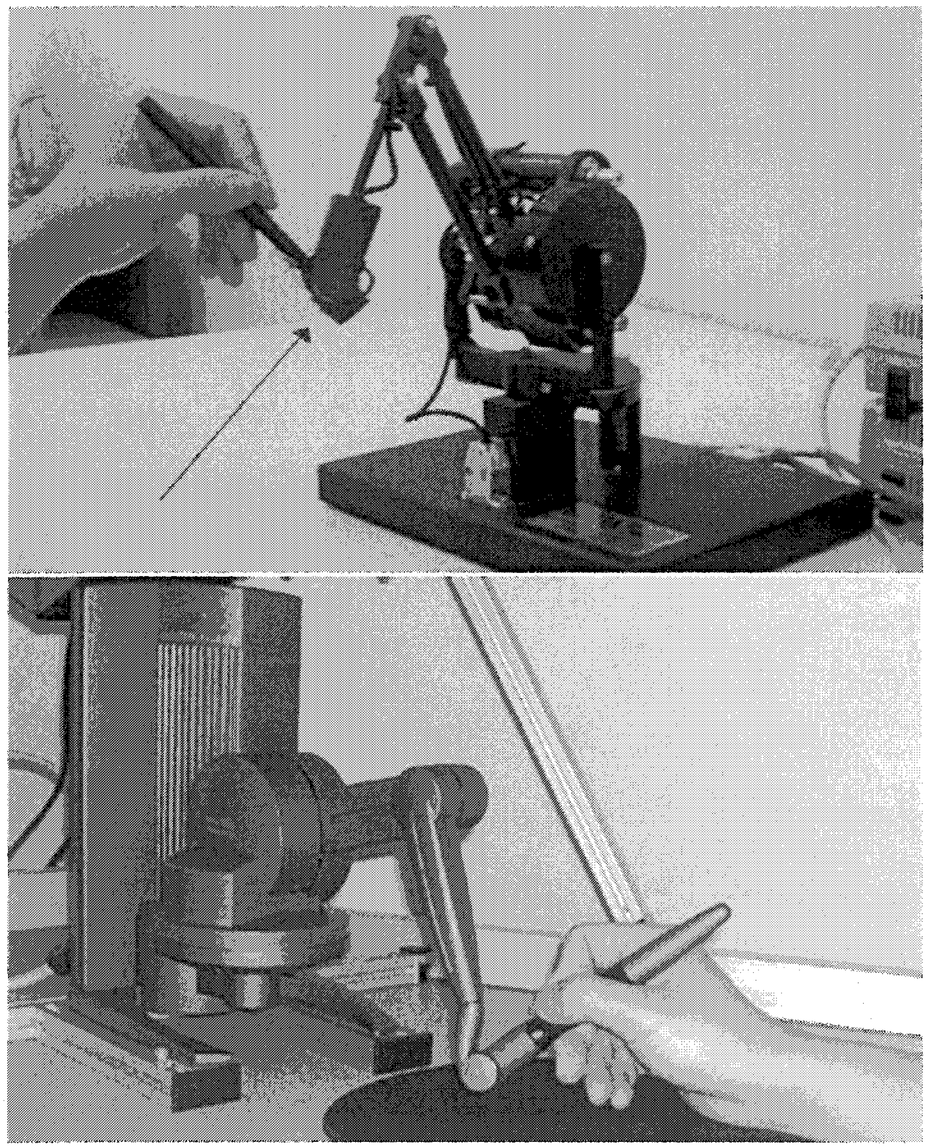

Figure 2.3: Two Different Models of PHANToM.

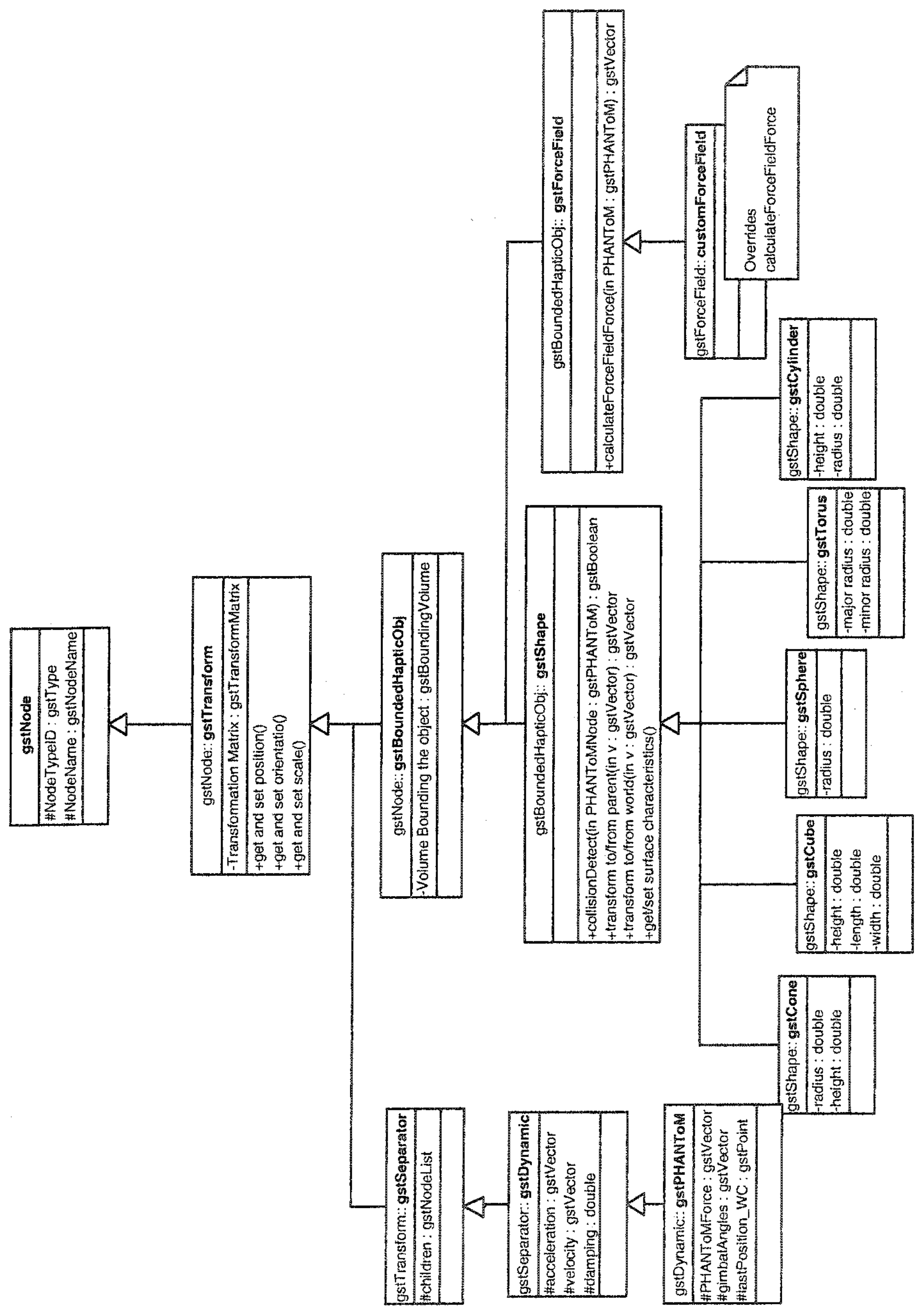

Figure 2.4: Partial UML representation of the GHOST API.

Classes gstScene and gstSeparator

The API allows the programmer to build a haptic virtual world with the same node mechanism as the one described in section 2.2.2, the role of the nodes being played by instances of gstSeparator. A Haptic scene contains a hierarchy of separators based on a root separator.

```
scene = new gstScene;
root = new gstSeparator;
scene->setRoot(root);
scene->startServoLoop();
```

Other separators and objects belonging to the scene will be added as children of the root. The public member startServoLoop() starts a new thread of execution that loops at a frequency of 1kHz. Every millisecond, the position of dynamic objects present in the scene is updated, callbacks are generated for the objects whose position or orientation has changed, and the force that has to be felt by the user is computed and applied to the PHANToM.

Geometry-related classes

The API provides a complete array of geometric structures that can be added to the haptic scene. Those structures inherit from the class gstTransform and can thus be translated or rotated by the calling application via the setRotate, rotate, setTranslate, translate, and setTransformMatrix methods. Transformations can be applied either directly to the objects themselves or to their parent separator.

Other objects that can be loaded into the haptic scene are shapes described by VRML files (function gstReadVRMLFile) and dynamic objects, whose position and orientation can be changed without any intervention from the calling application (gstButton, gstDial and gstSlider).

Class gstForceField

Force Fields can also be added to the scene, as a child to a gstSeparator. The developer has to define a new class for each kind of force field that has to be instantiated. It subclasses gstForceField and overrides the method calculateForceFieldForce. This method returns a gstVector, which will be added to the force applied to the PHANToM if the latter is inside the bounding volume of the force:

```
class magneticForce : public gstForceField {
   public:
   gstVector calculateForceFieldForce(gstPHANToM* phantom ,
                                       gstVector& torques){
      (...)
   }
}
```

Class gstPHANToM

Finally, gstPHANToM has to be instantiated and added as a child to the scene root. It is impossible to specify directly forces that have to be applied to the hardware. Those forces are computed and applied by GHOST according to the specified force fields and to reaction forces that are encountered when the PHANToM reaches a surface belonging to the haptic scene.

gstPHANToM contains methods that return the device's current position and orientation, as well as the reaction forces that are applied to it. The method *getPosition_WC* returns a point containing the position of the tip of the PHANToM in "World Coordinates", which means in a fixed referential whose origin is the position of the PHANToM base. *getGimbalAngles* returns the angles of the pen part of the PHANToM. Those angles are relative to the tip of the pen, and are not given in a World Coordinates system. Figure 2.5 shows a close-up of the gimbal angles at the joint of the pen part of the PHANToM and the arm that holds it.

getGimbalAngles returns a vector containing three angles, g_1, g_2 and g_3. These values can be used to transform the world coordinates system into a coordinate system fixed with respect to the PHANToM pen:

- $R_0 = (x_0, y_0, z_0)$ is the original fixed referential (world coordinates);
- $R_1 = (x_1, y_1, z_1)$ is defined by a rotation from g_1 around z_0;
- $R_2 = (x_2, y_2, z_2)$ is defined by a rotation from g_2 around y_1;
- $R_3 = (x_3, y_3, z_3)$ is defined by a rotation from g_3 around x_2.

Figure 2.5 shows how those angles are defined. If the same transformations are applied to an object in the graphic scene, its orientation follows the orientation of the haptic pen held by the user.

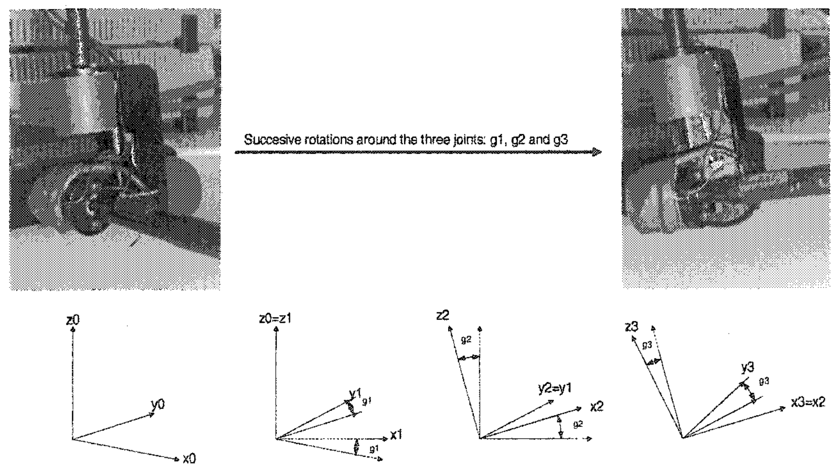

Figure 2.5: Gimbal Angles and Local Referentials.

2.4 Collaborative Virtual Environments: Going One Step Further than VE

The main contribution of this thesis is the improvement of communication between users of CVEs. This section details what a CVE exactly is, and the next section states the specific problems that appear in such environments.

The first step to build a CVE is to implement a *networked* virtual environment. Several design issues have to be faced [6]:

- Peer-to-peer or client-server communication architecture;
- Shared state management and data visibility;
- Packet compression and aggregation;
- Multithreading.

2.4.1 Communication Architecture and Shared State Management

The first two issues are linked: the client-server architecture consists of the description of the virtual universe and its components at one location (the server), which will receive and process updates from all the users, compute the new state of every object and send it back to all the users (the clients). Therefore, the server handles the state management. It is also quite simple to have the data visibility issue managed by the server. Since it knows where every user is situated in the virtual world, it can also decide whether a given user needs to be sent the state of a given object. For example, if the server knows that an avatar is looking in a given direction, it does not need to send it the state of an object situated behind it.

The other communication architecture, peer-to-peer, places the burden of state management on each user. No central description of the universe is available, and every user is responsible for keeping the global description accurate. Coherence, or synchronization, between the users (having every user seeing the same universe at the same time, with all the objects in the same position and state), is harder to maintain than in the previous architecture. [13] provides a review of the different strategies that may be used.

One way to solve the problem is to keep every shared object unique. If a user wants to see it or modify it, it has first to get the object through the network, and then process it, before eventually sending it elsewhere. The drawback of this approach is that the object as a whole has to go through the network for every modification.

Another possibility is to have one user (e.g. the user that created this object). responsible for the object's state management. Each time this object is modified, the repository user will send an update message to the other users. Since it is impossible to apply multiple modification requests concurrently, two strategies may be implemented to handle them:

- Serialize the incoming modification requests, queue them (either in a FIFO queue or in a priority-based queue), and process them one at a time;
- Implement an ownership management algorithm, and apply modification requests only if they come from the current owner.

The data visibility issue is tackled by setting *areas of interest* for each user. For example, in a 3D virtual environment, the user may specify a bounding box (with x, y and z coordinates) to be its area of interest. The user will then only be sent updates for object situated (at least in part) in this box.

The peer-to-peer architecture is more scalable than the client-server model (no server bottleneck), but is harder to implement. It also requires a multicast capability at the network layer, which is not needed in the client-server architecture. Multicast capability means that the sender of the message only needs

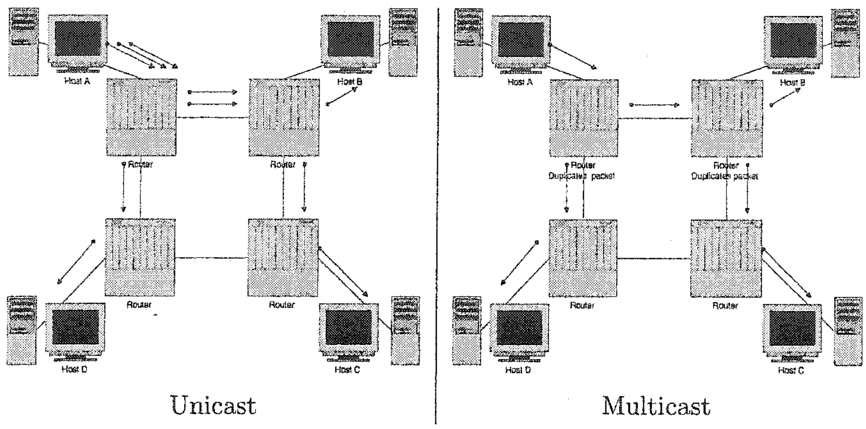

Figure 2.6: Unicast and Multicast.

to send one copy of this message, which will be replicated by the network itself for every destination, whereas unicast requires the sender to send one copy of the message for each destination (figure 2.6).

2.4.2 Packet aggregation and compression

Packet compression may be achieved by lossy or lossless compression algorithms. Lossless compression algorithms change the way data is encoded without losing any information (for instance, by using Run-Length Encoding). Another way is not to transmit an update, but differences between this update and a reference state. The required bandwidth is smaller in this case, but reference updates have to be transmitted reliably. Lossy compression is application dependant since the goal is to suppress "unimportant" data, which depends on the application itself. For audio or video streams, lossy compression algorithms have been well studied and have given way to formats such as MPEG1-Audio Layer 3 (Commonly referred to as MP3) for music, and MPEG2 for video compression [14, 15]. In the case of a haptic stream (the whole set of position, orientation, force and torque information that is exchanged in the course of a collaborative session), lossy compression could be done by replacing double-precision floating-point numbers by single precision floating-point numbers, thus dividing the size of update messages by two.

Packet aggregation consists in sending several updates in one network message. The overhead brought by network protocols is a constant number of bytes

m, therefore if n messages are sent as one packet, the total overhead is m *bytes* instead of $n * m$ *bytes*. Chapter 6, however, will show that such an approach is not efficient in Collaborative Virtual Environments.

2.4.3 Multithreaded architecture

The application needs to perform several tasks:

- Processing messages received from other users;
- Polling for the state of haptic devices;
- Computing the state of shared objects;
- Determining the new graphic representation and the new haptic scene;
- Eventually, sending updates to other users.

If all those tasks are done in a single thread of execution, they are constrained by the slowest one, usually the graphic rendering. For example, if the graphic rendering takes 99 milliseconds, and all the other tasks take 1 millisecond, those tasks will be executed only once every 100 milliseconds, which is far from enough for haptic devices, which require updates every few milliseconds.

With a multithreaded architecture, having several simultaneous loops is possible. One loop may then be created for every task, at a suitable rate of execution. However, special care needs to be taken to synchronize those tasks.

2.4.4 Cooperation vs. Collaboration

According to the definition given in [13], most distributed simulations merely offer cooperation: several user may interact and change the state of the same object, but not at the same time. Collaboration happens only when two users modify the state of the *same* shared object at the *same* time. Two users carrying a heavy object is a canonical example of collaboration. Both users need to lift the object at the same time, or else it will tilt and fall on its side, since one user alone can not carry it. The synchronization needed between the users is made particularly difficult by network impairments.

2.5 Effects of Network Impairments in CVEs

2.5.1 Delay and Jitter in CVEs

The effects of delay, jitter and packet loss in Collaborative Virtual Environments have been shown and partly quantized. [16] presents an experiment in which a collaborative task (bringing a ring from one end of a wire to the other, with one user moving the ring and the other one moving the wire) is performed using four network configurations:

- 10 milliseconds constant delay between users;
- Delay brought by an Ethernet LAN, ranging from 7 to 18 as an average;
- 200 milliseconds constant delay;
- ISDN Delay: average delay ranging from 150 to 300 milliseconds.

Two parameters were measured: the number of errors made during each trial, given by the number of times the ring and the wire entered into contact, and the completion time of the task. The same approach has been followed in this thesis, as it yields numerical evaluation of the user's performances.

Results of [16] have shown that a 10 milliseconds jitter is as bad, or even worse, in terms of collaboration, than a 200 milliseconds delay. No haptic devices were involved in this study, and there is no proof that the figures would remain the same in haptic virtual environments, but the results strongly suggests that having a slightly higher delay and cancelling the jitter would be beneficial. This is has been one of the main ideas behind the work presented in this thesis.

Another study takes only the delay into consideration [17]. 50 and 90 milliseconds latency were introduced. Those were device-induced, but the result would be the same for networked-induced latency, provided the total latency (or delay) reaches those figures, and that there is no jitter. The test consisted in having people in immersion walking in a virtual house and throwing a ball down a virtual pit. The users were given the illusion that the bottom of the pit was 6 meters below them. The stress they felt when throwing the ball was measured via physiological data (heart rate and changes in skin conductance brought by sweating), and a questionnaire after the trials. The study has shown that the extra 40 milliseconds delay have an impact on the feeling of immersion of the user. Users who experienced a 90 milliseconds delay were significantly less impressed by the pit than users who experienced the shorter delay.

2.5.2 Delay in Haptic CVEs

The preceding studies were made without the use of force feedback. The feedback helps the users to collaborate by transmitting the sense of touch in addition to the two senses that are commonly transmitted (sight and hearing). The drawback is that when the haptic device is part of a closed-control loop that encompasses the network, instabilities appear [18].

Helpfulness of haptic feedback has been shown in the context of two applications: removing the fuel tank of an aircraft with screwdrivers and wrenches [11], and identifying friends and foes on a monitor using a gear-shift like haptic device [18]. In both experiments, completion time with haptic feedback was roughly half the completion time without it. This advocates for the use of haptic devices in CVEs.

If the haptic device is controlled remotely, or if the haptic device controls another device over the network (which is the case in telesurgery applications), network delay has to be taken into consideration in the feedback brought by the control loop. A good review of teleoperation in presence of time delay for a single operator is done in [19]. [20] applies control theory and presents analogies with electrical circuits to close the control loop by monitoring only the slave device position, not its speed or forces applied to it.

These works address the issues of closed-loop control of haptic devices over networks that present delay. Our work, on the contrary, is focused on devices having local force feedback (the feedback loop itself is therefore not disrupted by the network delay), but with effects on a remote object.

2.5.3 Packet Loss in Haptic CVEs

All the update messages that are sent on the network do not have the same importance. For example, if a packet describing the position of a moving object is lost, the update message describing the next position of this object soon replaces the lost message. Therefore, this loss creates a small discontinuity in the object's position on the receiver side, which should not impede collaboration (unless of course bursts of update messages are lost). Therefore, the bulk of update messages can be sent unreliably. On the contrary, if the last position of an object is not properly sent, the receiver then has a wrong perception of the object's position and is not capable to interact with it.

Update messages that do need reliability (*key update messages*) also need to be received on time: the later a key update message is received, the longer the receiver has a wrong description of the updated object (skew time). More precisely, a key update messages can be defined as:

> An update message that represents the state of a shared object, such that the state does not change for a time equal to or greater than the maximum acceptable skew time [21].

Such update messages can be determined by the application based on implemented rules such as "the last update sent went a shared object stops moving is key" or "the update message immediately following a collision between the shared object and another object is key". The position of key update messages in a stream can also be approximated, by declaring update messages provided by the application as key with a given frequency (i.e. sending every n^{th} update message as key, or one key update message every t milliseconds).

This is why they need *timely reliability* [21]: the application has to make sure that those messages arrive, and that they arrive on time. Obviously, the higher the loss rate of the network is, the harder it is to meet this requirement.

2.5.4 Proposed solutions

New network protocols, like IPv6, allow the enforcement of Quality of Service (**QoS**), faster packet forwarding and a native support for multicast [22]. Together, those features may help decrease network impairments. However, delay will still be present, because of the finite propagation speed of the signals. Jitter and packet loss are likely to happen with any network protocol that is not connection-oriented (IP is not). Therefore, those impairments still need to be considered.

In cooperative telerobotics, several tactics have been proposed to make the users aware of communication delay [23, 24]:

- Enlarge the thickness of the local representation of the robot when this robot is moving;
- Display a predictive overlay: make an assumption on the actual robot position at the time the display is made, based on former positions and speeds, and display it on top of the delayed picture of the robot;
- Adapt the force to prevent collision when two robots are close one to another. Basically, this amounts to replacing the user input by a null force when the controlled robot is close to the other robot.

These tactics enable a faster cooperative manipulation of objects, with less collisions, in the presence of 500 milliseconds to 2 seconds delay. However, they are quite complex and application-dependant, and their efficiency has been proved only in presence of very high delays, with robots moving slowly manipulating objects with big inertia (the typical examples are underwater and space robots).

Another way to decrease the effect of the delay is to slightly alter the task [25]. For example, if the task is to bring an object into a box, displaying as an overlay the object already inside the goal (that is, displaying a picture of the task already completed) turns a placement task into a tracking task, which is easier to achieve for a human. Again, this approach is successful but application-dependant.

[26] proposes an architecture to smooth the jitter, at the cost of an increased total delay. In this architecture, update messages are sent by multicast to all the users taking part in the session. It is assumed that all the users share a synchronized clock. A good approximation of a synchronized clock may be reached by using Network Time Protocol (**NTP**) [27], or the Global Positioning System (**GPS**). A timestamp is attached to each update message that is sent. When this message is received by another host, its processing will depend only on its timestamp, not on the time it has actually been received. Note that this idea is not only a solution to jitter, but also to the delay awareness problem, since local actions are delayed as well as other actions (*delayed local display*). This is quite simple to implement and application independent, and therefore will be the approach followed in this thesis. The mechanism is explained in more details in chapter 3.

2.6 Transport Protocols

In this thesis, we tried to solve those problems at the transport layer. This section explains exactly what the concept of layer means, why we chose to work at this layer, and which solutions already exist.

2.6.1 Network Layered Architecture

Computer communication networks are considered and implemented as several *layers*, each layer providing services to the layer on top of it, and using services from the layer below it. The canonical architecture is the ISO's Open System

Interconnection (**OSI**) [28], shown in table 2.1.

Application	The computer application with which the user interacts.
Presentation	Platform independent representation of the data, so that applications running on different operating systems can communicate seamlessly.
Session	Session management: each user can join or leave a session, and send messages to the other members of the session.
Transport	Provides reliability, and causally ordered communications. While this is a requirement in file transfers, these features are far from being ideal in CVEs.
Network	Provides a set of logical addresses, that can be distributed in domains, making the routing task possible: if users are separated by several machines in a mesh configuration, the network layer tells the machines in between (the routers) where to forward the message so that it reaches its goal.
Data Link	Provides addressing, so that more than two users can share the same medium and still be able to send messages to a given host, and frame check, to ensure that the individual frame has not been corrupted by the physical layer.
Physical	The lowest layer is the one that actually carries the messages from one computer to another. Higher levels are abstractions, that provide a logical service to the user.

Table 2.1: The Open System Interconnection.

It should be noted that only the tree lowest layers are actually part of the physical network. The four highest layers are implemented by the end hosts only. When a message is sent from one user to another, it is passed from the highest to the lowest layer, sent on the physical link, arrives at intermediate nodes (routers), where it has to be processed (the message "goes up" to the network layer) and sent again, before it eventually reaches its destination, where it goes from the physical layer to the application layer, as shown in figure 2.7.

In this thesis, we will consider the three lowest layers as a given: their characteristics in terms of loss, delay and jitter are known, but cannot be changed by the users (for instance, IP without Quality of Service over Ethernet). The protocols used in the upper layers are agreed upon and implemented by the end

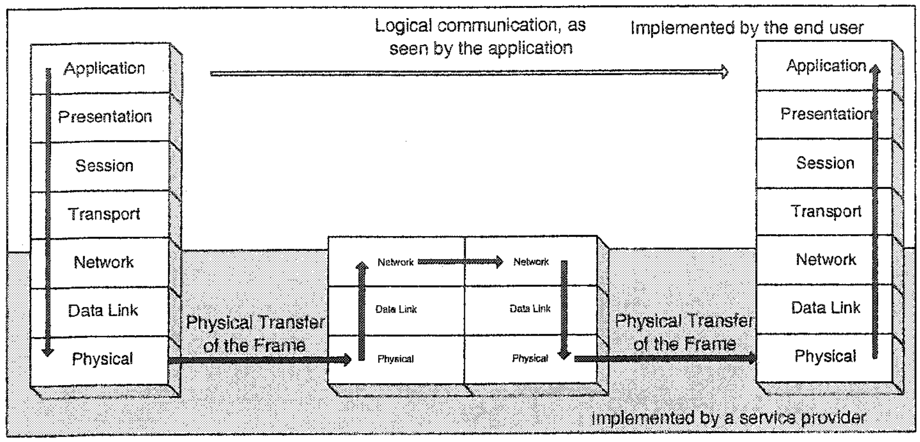

Figure 2.7: Message Exchanging in the OSI model.

users and can therefore be modified easily. This thesis focuses on the transport layer, since it can provide any application with the same service and therefore does not need to be implemented again with each new CVE application.

2.6.2 The two paradigms: TCP and UDP

On top of IP, two transport protocols have been developed: the Transmission Control Protocol (**TCP**) [29] and the Uniform Datagram Protocol (**UDP**) [30]. Both protocols divide the data to be transferred in packets, or datagrams, and use IP to transmit them from the sender to the receiver. TCP aims at doing this reliably, and at making sure that the packets arrive in the order they have been sent. This is important, for example, for the transfer of executable files, compressed archives or still pictures, since swapping parts of the data would make the whole file unreadable.

In TCP, packets to be sent are queued on the sender side. A sliding window mechanism is used to limit the number of packets that are sent at any given time. The size of the window is determined by congestion control considerations. When a packet is sent, a timer begins and in case there is a timeout, the packet is considered lost and sent again. On the receiver side, acknowledgments containing the packet sequence number are sent when a packet is received. The packets are queued and passed to the application according to their sequence number. For example, if packets numbered 1001 and 1002 arrive before packet number 1000, they are kept in a buffer. When packet number 1000 is received (assuming all the preceding packets have already been processed), packet number 1000, then number 1001, then number 1002 are passed to the application. This ensures a *causally* and *totally ordered* communication. Such communications are not suitable (or even necessary) for distributed real-time applications [31]. If update messages are sent over TCP, they will be displayed by the receiver in the order they have been sent. However, if one update message is lost, and the following updates are received, the latter will be kept in a buffer, not being passed to the application. TCP will do everything to retrieve an update message that is already obsolete (since subsequent ones are already received), which is undesirable by itself since valuable network resources are used without any gain. Moreover, queued update messages received in time may well become obsolete themselves before they reach the application, which will significantly impede collaboration.

The other alternative, UDP, does not provide any reliability. It is a very simple protocol, which only gives port numbers (to have several applications using the network at the same time on a single IP address) and checksum (to ensure that the packet does not contain bit errors). No attempt is made to recover lost packets, or even to detect that packets have been lost. While this protocol does not present the drawbacks of TCP and offers timely processing (no buffering happens), it does not suit the reliability requirements of collaboration.

2.6.3 Other transport protocols

A TCP-like protocol ("Light TCP")

An *updatable queue* can be implemented into a transport protocol adapted to CVEs that present certain similarities with TCP [32]. To support the concept of *message obsolescence*, the updatable queue (on the sender side), accepts any update message coming from the application and processes them as follows:

- If the message is a key update, it is placed at the end of the queue, and a marker is placed after it in the queue, to prevent it from being erased;

- If the message is a normal update, it can replace older updates for the same shared object (The older update message being now obsolete does not need to be transmitted). However, replacing an older update message with the one that has just been received from the application is possible only if no update message is present between the position of the older message and the end of the queue. If such is the case, the new message will simply be added at the end of the queue.

- Update messages that have already been sent but which have failed to be acknowledged are placed again in the queue if no more recent messages for the same index are already present.

Messages are taken from the queue and sent as one IP packet (message bundling), the total size of the packet being determined by a congestion control algorithm (the more congested the network, the smaller the number of updates transmitted at the same time).

At the receiver side, update messages received are forwarded to the application only if they are more recent than what has already been received. There is no buffering.

The similarities with TCP are:

- A queuing of the messages to be sent (However the queue is not a FIFO queue);
- A per-packet acknowledgment mechanism, each packet being sent again if no acknowledgment is received after a certain time;
- The usage of sequence numbers;
- The sliding window and congestion control algorithm.

[32] does not give any name to this protocol. For convenience, it is called "Light TCP" in the remainder of this thesis.

SCTP

The Synchronous Collaboration Transport Protocol (**SCTP**)[2] has been designed specifically for update message transmission in CVEs [33]. Normal (non-key) update messages are sent unreliably. The protocol is similar to UDP, except for a sequence number used to order messages at the receiver (same thing than in the TCP-like protocol). Key updates are sent reliably: when they are sent, a timer is triggered. If a timeout happens before an acknowledgment is received,

[2]In this work, SCTP *does not* stand for the Stream Control Transmission Protocol. The name of the protocol described here was chosen before the stream control transmission protocol became a well-known general-purpose transport protocol. Since no other name has been provided for the synchronous collaboration transport protocol, and knowing that the context is unambiguous, we chose to keep this abbreviation.

the message is sent again, still as a key update message. It has been proven than for collaborative applications, SCTP, which is acknowledgment-based, performs better (in terms of errors happening during a collaborative session) than protocols based on negative acknowledgments, in which the receiver has to detect a loss and ask for a retransmission. [33] does not take jitter (either generated by the network or created by the retransmission of update messages) into account.

RTP/I

The Real-Time Application Level Protocol for Distributed Interactive Media (**RTP/I**) [34] has been developed as a logical extension of the Real-Time Transport Protocol (**RTP**). RTP itself provides sequence numbers (to determine if a received update message is more recent than what has already been received or not), timestamps, session support (the ability to know who is using the virtual environment, in terms of IP addresses, the ability to resynchronize its own view from the world with others, the ability to get a description from this world being a latecomer), and session control (the Real Time Control Protocol (**RTCP**) provides primitives as "play", "pause", "fast forward" and so on). RTP/I itself adds packet content identification (to distinguish state messages, update messages and state queries), subcomponent identification and sequence numbers, so that different subcomponents of a same component do not need to be updated if only one subcomponent changes. RTCP/I adds further subcomponent support and name mapping.

This protocol is useful for applications such as shared whiteboard and teleteaching, but does not offer any support for key updates. Furthermore, its features like state packet and state query packet would not be used in our framework, this is why it has not been further considered in this thesis.

2.6.4 Reliability and Scalability

In early works on distributed virtual environments, and particularly in distributed simulations, scalability (ensuring that the application could support an arbitrarily big number of users) has been one of the main concerns. There has been a shift in this trend, as virtual reality community came to understand that for a virtual universe it was pointless to worry about supporting tens of thousands of users if this universe did not succeed in interesting more than ten individuals at the same time. Until recently, one of the main questions in surveys and taxonomies about virtual environments was: "Is it scalable?" [35].

This has prompted researches in *reliable multicast* and *large scale multicast applications*. Such applications require hybrid protocols featuring both best-effort and reliable multicast [36]. In distributed simulations, five different transmission modes have been identified:

- Best-effort, low-latency multicast for update messages transmitted at a high rate;
- Low-latency, reliable multicast for objects which may transmit update messages at arbitrary times;
- Low-latency, reliable unicast for occasional communication between a few number of hosts, in the case of a collision for example;
- Reliable multicast distribution for scene descriptions;
- Reliable unicast for control information.

[36], however, does not propose any implementation for such protocols.

The Selectively Reliable Transport Protocol (**SRTP**) is one implementation of reliable multicast [37]. SRTP combines a best-effort protocol with a reliable multicast mechanism, which is based on negative acknowledgments. The distinction is not made between key update messages and normal update messages, but rather between rarely-changing and frequently-changing data. The assumption is that if a packet describing frequently-changing data is lost, another description will rapidly follow and the loss will remain unnoticed.

Dual-mode multicast has been proposed to achieve scalability [38]. "Dual-mode" means that there are two layers of multicasting: one for each Local Area Network (**LAN**), and an underlying layer for the Wide Area Network (**WAN**) connecting the LANs. At each LAN an agent filters the data multicasted on the LAN and decides whether packets have to be forwarded to the WAN or not.

However, there is not really any concern for scalability in haptic CVEs. While thousands of users may be taking part into a simulation, there will never be more than ten, or possibly twenty users engaged in a given collaboration, like repairing a damaged tank or performing surgery on a patient.

2.6.5 Multi-protocols Architectures

The ability to switch between several protocols, one for large-scale distributed simulations and one adapted to collaboration when several users meet and need

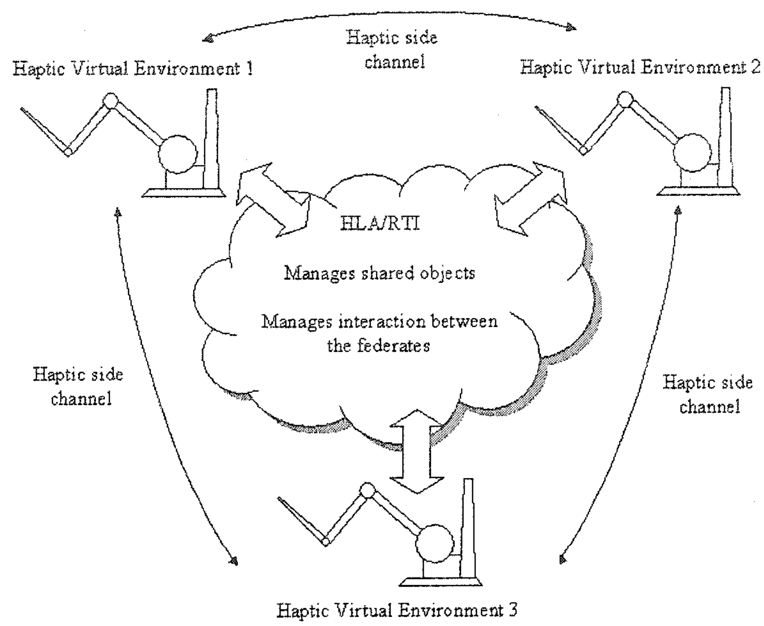

Figure 2.8: Multi-Protocol Architecture for Haptic CVEs [39].

to perform a collaborative task, for example, requires an architecture that supports those different protocols. Haptic Collaboration Group Management can be done through HLA object declaration and Data Distribution Management [39].

HLA (implemented by the Run-Time Infrastructure) provides methods to match data producers and consumers. The format of the data is transparent for HLA/RTI, and the way this matching is done is transparent to the users. Therefore, HLA/RTI can be used to exchange haptic information between federates (users in a same federation, or users of a common virtual universe). Once two federates have found enough data about each other, they can switch to a side-channel, using a collaboration-oriented protocol to carry haptic update messages.

Chapter 3

Protocols Comparison

This chapter presents the three transport protocols that have been implemented in this research. SCTP relies on the acknowledgment of key updates, the sender retransmitting those updates if no acknowledgment is received. This is opposed to the negative acknowledgment approach, where the receiver has to detect the loss and send a request for a retransmission (e.g. in SRTP). This ensures *timely reliability of key updates* over networks where loss is present [21].

Yet, if jitter is present in the network, SCTP does not propose any solution to remove it or smooth it out. This is why, during the course of this research, Smoothed SCTP has been devised and implemented. Smoothed SCTP implements the same acknowledgment policy as SCTP, and adds buffering on the receiver side to smooth out jitter.

Finally, "Light TCP" has been considered for its aggregation capability, which could lower the bandwidth requirements. The latter protocol queues the update messages on the sender side, so that obsolete messages can be pruned before they are sent, and other messages can be bundled in a single network packet.

3.1 SCTP

When an update message is received from the application and has to be sent over the network, its index (given in the update message to transmit) is retrieved.

A sequence number, equal to the maximum value of received and sent sequence numbers for this index plus one, is then appended to this update message. This choice is coherent with the assumption that only one user at a time sends update messages for a given shared object (see section 3.5.3). In multicast mode, one sequence number is kept for each index, and in unicast mode, there is one sequence number for each (index,destination) couple. That means that for the same index, the sequence numbers for update messages sent to two different destinations are independent.

Then the packet is built:

Header	Index	Double	Double	...

Update Message with 6 DOF

Header:	0	Sequence number	Key Status

Table 3.1: SCTP Packet Format.

The first field, 0, indicates that it is an update message. 1 will be used for acknowledgments.

If the update message is a key update, a timer associated with the processed index (and in the unicast case, with the destination ID as well) is created, and any timer already associated to those parameters is cancelled, to prevent a timeout when a more recent key update has already been sent. The applied timeout (time after which an unacknowledged update message will be considered lost) is a global variable that can be set by the application. A usual value is two times the average packet round trip time (in case there are more than two hosts, the maximum average round trip time is considered). The sent update message, its index and its sequence number are stored in the timer thread, as well as the number of hosts this update has been sent to. The latter is one in unicast mode, and given by the session server in the multicast mode. When an acknowledgment is received for an update message, the corresponding timer is cancelled (in unicast), or has its counter decreased by one and cancelled when it reaches zero (in multicast). If the timer is not cancelled before its timeout, a new update message is generated and added to the message stream, with its original sequence number.

On the reception side, a new thread is launched when the class is instantiated. This thread receives and processes one packet from the network layer in every loop. If the packet is an acknowledgment (begins with the byte 1), the corresponding timer is cancelled, as explained in the previous paragraph. If the packet contains data (begins with the byte 0), the sequence number, key status and the update message contained in the packet are extracted. The sequence number is compared to the last received sequence number for this index, and the update message is forwarded to the application (via the *updateReceived* callback) only if it has a higher sequence number than the sequence number already stored.

If a key update is received, an acknowledgment is generated. The received

1	Sequence number	Key Status	Index

Table 3.2: SCTP Acknowledgment.

sequence number is copied into the acknowledgment, as well as the index of the received update. The field indicating that it is a key update is kept. This is actually useless for the processing of the update message, but simplifies the implementation.

The finite state machine representation of the SCTP protocol can be deduced from those specifications, and is shown in figure 3.1.

SCTP is a protocol that offers fast processing for common update messages and timely reliability for key update messages, in a network that present packet loss or delays. However, it does not address the issue of jitter, which is why Smoothed SCTP has been devised.

3.2 Smoothed SCTP

Smoothed SCTP is an extension to SCTP aimed at reducing the network jitter and providing delay awareness to the users. The communication process is the same as in SCTP. The only difference in the sending mechanism is that a timestamp is added in the header. This timestamp is given by adding to the host current time the time shift that exists between the host clock and a common NTP server. This shift is calculated at the beginning of the session by querying the NTP server. It is assumed that the host local clock will remain accurate during the session, which means that the shift between the local clock and the NTP server remains constant, and that $Local\ time + Initial\ shift$ always gives

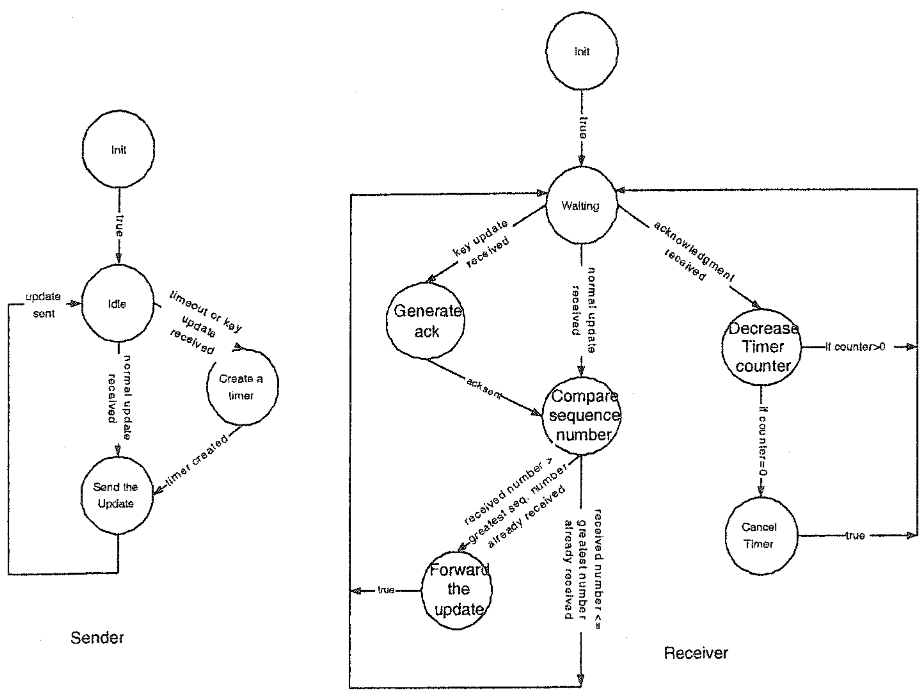

Figure 3.1: Finite State Machine Specification of SCTP.

the synchronized time from the NTP server. All the times mentioned here are given in milliseconds.

The packet header has the format described in table 3.3. This header can actually be used for both SCTP and Smoothed SCTP, the former then simply discarding the timestamps from the received messages.

0	Sequence number	Key Status	Timestamp (long integer)

Table 3.3: Smoothed SCTP Header.

Another difference between SCTP and Smoothed SCTP is that when the protocol is instantiated, a "delayed protocol" flag is set in the toast.transport.Comm superclass. This causes every update sending by the application to call the *sendLocalUpdate* method as well as *sendUpdate*. The update message is then handled as if it had been received from another host.

When an update message is received, it is extracted from the network packet, and acknowledged if needed. But instead of being forwarded directly to the application, it is added to a bucket depending on the timestamp it carries. This bucket will be the same no matter what the arrival time of the packet is.

On the receiver side, arriving update messages are put in buckets. Two parameters have to be chosen by the application designer:

- The bucket length, B_L, given in milliseconds. The smaller the bucket size, the smaller the residual jitter, but also the greater the computational load of the protocol;
- The delay, expressed in numbers of bucket length, $\Delta = n * B_L$.

Those two parameters may change from one application to another, but have to be the same for all the users of a shared virtual environment using this protocol.

When an update message with timestamp t is received, it is added to the bucket number $\left\lfloor \frac{t}{B_L} \right\rfloor$. For instance, if $t = 12345678\ milliseconds$, and for $B_L = 10\ milliseconds$, the update message is put in bucket number $\lfloor \frac{12345678}{10} \rfloor =$ 1234567.

Every $B_L\ ms$ the receiver checks the bucket corresponding to updates that have been sent $\Delta\ ms$ ago. If it finds any updates in this bucket, it forwards them to the application. In our example, let us assume we have chosen

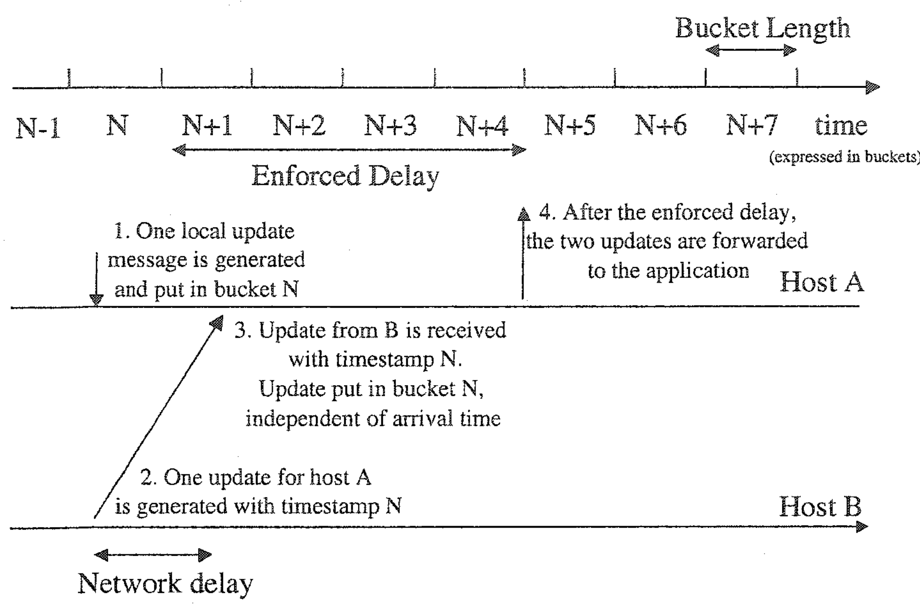

Figure 3.2: Bucket-Based Synchronization.

$\Delta = 4 * B_L = 40\ ms$. At time 12345720 ms, the receiver checks the bucket $\frac{\text{Current Time}-\text{Enforced Delay}}{\text{Bucket Length}} - 1 = \frac{12345720-\Delta}{B_L} - 1 = 1234567$, finds and forwards all the updates that have been received with a timestamp between 12345670 and 12345679,[1] thus including the update message mentioned above.

The host also applies the bucket mechanism to its *own update messages*, which means that even if they have been produced locally, those updates will be passed to the application only after Δ ms.

The average delay experienced by the users is $\Delta + \frac{B_L}{2} + \Delta_{App}$ where Δ_{App} is the delay brought by the application layer, and the maximum jitter is $B_L - 1\ ms$, and is independent from the original jitter. It should be noted that this approach may increase the packet loss rate, since an update message arriving with a delay greater than or equal to $\Delta + B_L$ is placed in a bucket that will never be considered again, which is equivalent to it being discarded. This may happen when Δ is too small compared to the network delay, or when unusual and unexpected jitter happens.

After the *enforced delay*, each bucket is read and the update messages found

[1] The term "-1" in the formula comes from the fact that buckets are polled at the end of a time interval, whereas their number reflects the time at which they began.

into it are forwarded to the application. Not only does it reduces the jitter, it also smoothes out update bursts: if for a reason or another an important number of updates are received in the same bucket for the same shared object, only one (the most recent) is forwarded to the application.

This protocol also brings delay awareness to the user, since the result of their own actions is delayed by the enforced delay. The synchronization between the users actions is then made easier, hence the collaboration is easier as well. The cost of this added delay is a potentially reduced feeling of immersion.

Another problem that arises is that when an key update message is lost because it experienced a delay greater than Δ. In this case the update is acknowledged (key update), but is nevertheless lost. Passing this message to the application anyway would reintroduce jitter, which is something to avoid. Such an update message, even if key, is probably outdated since it has been delayed by an unusual amount of time. The issue has not been addressed in this work, but needs attention, especially if the update that is lost because of an unusual delay is the last one in the stream (in which case it is vital that it is forwarded to the application, to keep consistency among the users).

3.3 Light TCP

3.3.1 General Idea

"Light TCP", an attempt to modify TCP specification for collaborative environments, has been called this way because of its similarity with TCP. Many non-essential features of the original protocol have been removed. Moreover, the concept of updatable queue has been implemented, instead of First In First Out (**FIFO**) queues. Rather than queuing the update messages at the receiver side, as done in smoothed SCTP, the queuing is done at the sender side. This can smooth the interaction stream of update messages (though not as efficiently as in the previous protocol, since the network jitter cannot be taken into account), and by preventing the transmission of useless update messages, helps reducing the required bandwidth. Furthermore, this protocol bundles update messages together, which saves overhead and further reduces the amount of transmitted data.

3.3.2 The Updatable Queue

A different sending queue will be managed for each destination. That leads to the use of one different thread for each destination in unicast. In multicast, there is only one destination (the multicast group), therefore only one thread is needed to manage the queue of update messages to send.

The queue being adaptable, when an update message arrives it replaces the last update message already in the queue with the same index. However, key update messages cannot be overwritten. To mark key update messages, a special update message is introduced in the queue. It bears the index -1, so that no update message sent by the application can be mistaken for this marker.

The algorithm becomes the following, with the notion of order ("before", "after", "last") corresponding to the order in which the updates have been added:

- If a key update is received from the application, place it after any existing update, regardless of what is already inside the queue. Then add a special update message with index -1 in the queue;
- If a normal update is received, check the position of the last update message already in the queue bearing the same index. If such an update message does not exist, or if this update message is placed before a special -1 update message, add the received update message at the end of the queue. Otherwise, overwrite the older update message.

Finally, another special update message, with the index -2, has to be introduced to manage timeouts. If an update message is sent on the network, and fails to be acknowledged (see below for the timer management), it is added to the sending queue again, unless it is obsolete. When a timeout is generated, the following algorithm is used:

- If no update message with the same index is already present in the list, add the update message issued by the timeout. Then add a special update message with index -2;
- If the sending queue already contains an update message with the same index, and if this update message is immediately followed by the -2 special update message, replace the older update message;
- If none of those two cases apply, drop the update message, a more recent one is already in the queue.

The numbers represent the index of the update messages, and the letters give the order in which those updates have been generated.

Update Message 1_A was in the queue and has been sent.

							$\rightarrow$	1_A

Update Messages 1_B, 2_A and 3_A have been received from the application.

				3_A	2_A	1_B

Update Message 1_C is received from the application.

				3_A	2_A	1_C

Key Update Message 2_B is received from the application.

		key	2_B	3_A	2_A	1_C

Update Message 1_A has not been acknowledged. A timeout happens. 1_C is already in the queue so 1_A is discarded.

1_D is received from the application.

	1_D	**key**	2_B	3_A	2_A	1_C

Then the network layer is available for sending, the queue is flushed and the update messages are sent in this order: 1_C,2_A,3_A,2_B,1_D.

Figure 3.3: The Updatable Queue: An Example.

The second case is justified by the fact that the update message already in the queue has itself been added after a timeout. Since the timeout that brought the update message already in the queue happened before the timeout currently being processed, the update message already in the queue is older than the processed update message. This is why the new one overwrites the one already in the queue.

This updatable queue management is illustrated in figure 3.3

3.3.3 Emission Mechanism

When the emission thread is ready to send a packet (after the preceding packet has been processed), it checks the queue of update messages to be sent. To avoid busy waiting, the thread is suspended when no update message is present in the queue. A new packet is created, its length equal to the minimum of

Header	Update Message	Update Message	...

Header:	Timestamp	$N_{Updates}$

Table 3.4: Light TCP Format.

$maxPacketLength$ and $5 + N_{queue} * UpdateSize$, where N_{queue} is the number of update messages currently present in the queue. $maxPacketLength$ can be changed dynamically to provide congestion control and constrain the packets to a smaller size when the network is overloaded. However, $maxPacketLength$ has been kept constant in the implementation presented in this thesis, with the assumption that collaboration would be impossible with overloaded networks, no matter how good the transport protocol is. If $maxPacketLength < 5 + N_{queue} * UpdateSize$, the queue cannot be emptied and some update messages will have to be sent in a subsequent packet.

The packet is built according to the structure shown in table 3.4.

The timestamp is used for the management of the acknowledgments, and does not need to rely on a synchronized clock. The time used is the current time given by the local clock (in milliseconds) from which the application starting time is subtracted. This way, the initial values of the timestamp are quite small, and four bytes are enough to code this timestamp. The maximum value that can be reached before a buffer overflow is $2^{31} - 1 \approx 2.1 * 10^9$ $milliseconds$, which represents more than 24 days after the session has started. The next byte is used to give the number of update messages contained in the packet. This number is actually set after the update messages have been added. To fill the packet, update messages are removed one by one, starting at the beginning of the queue. Special updates with index -1 or -2 are used for the queue management only, and do not need to be sent. If such an update is taken from the queue, it is discarded and the next one is retrieved from the queue. The field "number of updates" is then set, and if the packet initially allocated is longer than what has to be sent, the packet is trimmed. Finally, it is sent to the network layer.

3.3.4 Acknowledgment Policy

For every update message put in the packet, a timer is started. A table keeps track of the latest timestamp sent for each index. In unicast mode, this needs to be done separately for each destination. When using multicast, the number of destinations at the time the packet is sent is kept in another table. If a timer was previously associated with the index of a sent update message (as a result of a former sent update message), it is cancelled. This means that for each shared object, only the latest update message needs to be acknowledged.

When an acknowledgment is received, each index it contains is processed only if its timestamp is equal to the latest timestamp sent for the index. In this case, the corresponding timer is cancelled (unicast mode), or the number of acknowledgments to wait is decreased by one (multicast mode) and the timer subsequently cancelled when this number reaches zero.

If a timer is not cancelled before it times out the corresponding update message is processed and possibly added to the sending queue again, according to the policy mentioned above.

The exact value of the timeout is computed dynamically. The initial value of the timeout is 200 milliseconds, a delay above which collaborative work seems hard to achieve [16]. For each received acknowledgment, the timeout is adapted, using the Jacobson's and Karn's algorithms [41, 42], which are also applied in TCP. The variables used are the Round Trip Time (**RTT**), which is the time needed for a packet to go from the sender to the receiver and back, and an additional delay that is added not to have an excessively small timeout that would detect a loss when the packet is only delayed by more than the RTT. The algorithm is the following:

- Compute $diff$, the difference between the current time and the timestamp of the acknowledgment (which is the time at which the original packet has been sent);
- Make $error \leftarrow diff - RTT$;
- Update RTT: $RTT \leftarrow RTT + error/8$;
- Update the additional delay: $delay \leftarrow delay + \frac{|error| - delay}{4}$;
- Update timeout: $timeout \leftarrow RTT + 4 * delay$.

3.3.5 Reception Mechanism

When a bundle of update messages is received, an acknowledgment is prepared and sent. This acknowledgment bears the timestamp of the original packet and a negative number of update messages (to differentiate the acknowledgment packet from a data packet), the absolute value of this number being the number

TS	N	Sender ID	Ix	Ix	...

TS: Timestamp
N: Number of acknowledged updates (Negative number)
Ix: Index of the acknowledged update,
in the order of the original packet

Table 3.5: Light TCP Acknowledgment.

of updates indicated by the received packet. The subsequent String represents the ID of the data packet's sender. This is required in multicast mode, since acknowledgments will be sent to the multicast group and this field is the only way for a given host to determine whether a received acknowledgment corresponds to a packet it has sent or not. Then the index numbers are copies of the original indexes contained in the packet, in the same order. If the update messages are more recent than the last ones already received (this can be determined through the received timestamp), they are forwarded to the application.

If an acknowledgment is received, and if this acknowledgment bears the ID of the host that received it, it is processed according to the acknowledgment policy.

3.4 Protocols Theoretical Comparison: a Trade-off between Reliability and Timely Processing

The chart 3.6 gives a summary of the comparative features of the three considered protocols, as well as UDP and TCP. All three protocols are based on (positive) acknowledgments. However, it should be noted that they do not rely on acknowledgments for the same purpose. In SCTP and smoothed SCTP, they are used to provide selective reliability for key updates. In Light TCP, the last bundle of update messages that has been sent is acknowledged. The "key updates" concept is supported in the way the sending queue is managed. The added value of smoothed SCTP(jitter smoothing) and Light TCP(reduced bandwidth requirement) both come at the price of delayed processing (the update messages are no more sent as soon as they are produced by the application AND forwarded to the application as soon as they are received).

	FP	KU	AB	JS	Agg
UDP	✓	×	×	×	×
SCTP	✓	✓	✓	×	×
Smoothed SCTP	×	✓	✓	✓	×
Light TCP	×	✓	✓	×	✓
TCP	×	×	✓	×	✓

More timely ↕ More reliable

FP: Fast Processing
KU: Supports Key Updates
AB: Based on Acknowledgment
JS: Jitter Smoothing
Agg: Enables aggregation

Table 3.6: Protocols Comparison.

These protocols have been ordered on the table according to their relative position in the tradeoff between more timely protocols, and more reliable protocols. [21] has already demonstrated that SCTP was better than less reliable (and particularly less *timely* reliable) protocols. In the remainder of this thesis, a practical evaluation of those protocols is proposed to find the optimum for collaborative work using haptic devices.

3.4.1 Out of Synchronization Time

The Out of Synchronization time (**OOS**) is the time during which the user does not have a consistent description of the universe. For the tracheostomy application, this amounts to not having the same position of the tools or state of the skin as the other users, and for the box carrying application, it means to have a local position of the box different from the box position given by the server.

In smoothed SCTP, provided that lost key updates can be retransmitted

Method	Parameters
sendUnicastUpdate	Update message to send, Key status of the update, String representing the ID of the destination.
sendMulticastUpdate	Update message to send and its key status.
setTimeout	Timeout to apply.
flushHostInfo	Destination for which internal data has to be flushed.
stop	

Table 3.7: Protocols Interface with the Application.

Method	Parameters
sendPacket	Packet to send (array of bytes).
sendPacket	Packet to send and destination.
receivePacket	Packet to receive (same approach as the Java socket implementation). This method returns a String representing the ID of the packet's sender.

Table 3.8: Protocols Interface with the Network Layer.

before the bucket corresponding to the original update is read at the receiver side, this OOS is equal to the enforced delay. This is a heavy price to pay, since SCTP can achieve smaller OOS [21] but has the advantage of yielding a known, constant OOS.

3.5 Common Implementation issues

3.5.1 Interface

The protocols' interface with the application is given by table 3.7. Their interface with the network layer is given in table 3.8.

The multiplexing (Sharing the same IP address between different applications that run on the same computer and need to access the network) and packet checksum (to ensure that the packet has not been modified by a faulty lower-layer transmission) are directly handled by the Java implementation of UDP, and not by the protocols presented here.

3.5.2 Unicast and Multicast

Every transport protocol supports underlying multicast and unicast transport. Theoretically, the transport layer is independent from the network layer, and therefore the fact that the latter uses unicast or multicast should not make any difference to the former. However, this is not the case in reality. For clarity, only IP is considered here, but the statements remain valid for any protocol that makes the distinction between unicast and multicast. Using unicast, any datagram is given a destination address (at the network layer) and a port number (at the transport level). IP then forwards the packet to its destination address, and when this packet is received at the host, it is passed to the application listening on the destination socket. In multicast, every application wishing to receive update messages for a given session registers itself in a *multicast group*. This group is an IP address in the range 224.0.0.0 to 239.255.255.255 (224.0.0.0/4, class D address). This address, and the port used, have to be agreed upon by the users before the start of the session. All the update messages are then sent to this multicast address and port, and forwarded to all the hosts registered on this IP address.

Using unicast or multicast brings three changes in the transport layer:

- In unicast, the destination address has to be known by the sender. This is not the case in multicast;
- In unicast, it is possible to send again an update message to a destination host that failed to receive it. This is not possible any more in multicast, and if one host fails to receive an update message, it has to be sent again to the whole multicast group;
- In multicast, acknowledgments are sent to every host, just as update messages are. Any host waiting for an acknowledgment must ensure that an acknowledgment it receives correspond to a message he sent, and not to a message sent by another host.

For those reasons, the use of a multicast or a unicast network layer will be handled differently in the transport protocol.

3.5.3 Concurrent Sending of Update Messages

A key assumption has been made: *only one host at a time sends update messages relative to a shared object.* This is the cost of working independently from the application: no dead reckoning or estimation of the updates messages sent by other hosts being available, it is impossible to estimate what will be done in the future. Therefore, it is impossible to keep a synchronized description of all the shared objects across the different hosts if more than one host modify the same object at any given time. The application *must* be designed with this constraint in mind. Two solutions are possible:

- Implementing ownership management for the shared objects so that only the current owner of an object sends update messages for this object. This is the approach followed for the tracheostomy application;

- Implementing a client-server application, in which only the server sends update messages for collaboratively operated objects. The clients send update messages as well, but only for shared object wohse state does not have a direct influence on the other client's objects. For example, in the box carrying application, there are three shared objects: one box and two hooks. Each client sends the position of its hook to the server, which in return sends the position of the cube to both clients.

Chapter 4

Comparison Framework

The comparison framework developed in this thesis has been called Transport Layer Optimization for Applications in Surgery and Training (**TOAST**). Its goal is to provide a complete interface to both applications which require haptic collaboration and to transport layer protocols. Its initial design has been inspired from INVENTIST [21].

4.1 Requirements

The Framework has to provide:

- A precise format for the update messages, that can easily be modified if the application's requirements change;
- Management of the Interaction Stream;[1]
- An interface to both the transport protocols (protocols are specified in chapter 3) and to the application. Those interfaces are used to exchange update messages;
- A network layer impairments simulator;
- Logging of exchanged update messages, and application-specific events. For each user, incoming and outgoing update messages have to be logged, along with the time at which the event happened. This way, subsequent analysis of the session is possible;

[1]The Interaction Stream is the stream of all updates sent and received by the users, made up of regular update messages and key update messages.

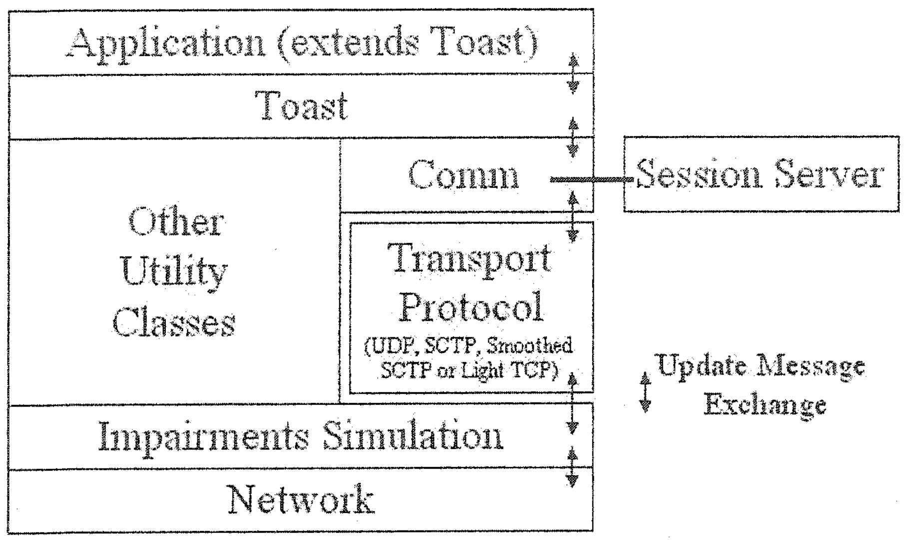

Figure 4.1: TOAST Architecture.

- Logging amount of network traffic generated;
- Reliable session support, to handle the number of users connected at any time.

4.2 Architecture

The framework has been divided into three packages (see figure 4.1):

- *toast*, which provides the main class of the framework and the update messages format;
- *toast.util*, that gathers several useful classes;
- *toast.transport*, where the transport protocols are implemented.

A complete list of the classes implemented in those packages is given in appendix B.

4.2.1 Application Interface

The interface to the application is provided by *toast.Toast*. Any application using TOAST has to extend it to be able to register shared objects of the virtual environment and exchange update messages with it. Doing so, the application inherits the methods *sendUpdate* and *sendKeyUpdate*, and is required to implement the abstract method *receivedUpdate*, which is a callback triggered by the transport protocol when passing an update message to the application.

The interaction stream can be managed either by the application ("manual" handling), or by TOAST ("automatic" handling). In the former case, the application specifies whether the update message to transmit is a key one or a normal one, and uses sendKeyUpdate or sendUpdate accordingly. In the latter, the application always uses sendUpdate and TOAST determines which updates among those are to be sent as key updates. This is done by sending every n^{th} packet as key, or sending one key update every t millisecond, n and t being parameters given by the application.

Finally, the class Toast also logs the update messages when they are sent or received. A timestamp and a copy of the update message is written in a log file at each exchange.

4.2.2 The Update Messages

The format of the update messages is given in the class toast.UpdateMessage. Each time a new update message is generated by the application or read from a network packet, a new instance of this class is created. Update Messages are composed of one index, that identifies which shared object the update message applies to, followed by DOF double-precision floating point numbers, DOF being a number specified by the framework. This number is constant throughout a session, but can be changed if the need arises. It has been set to six, so that the position of a shared object can be fully described (three coordinates for translation and three coordinates for rotation). Another alternative is to exchange only the position of the object in translation, as well as the forces applied to it.

4.2.3 Transport Protocols Interface

The transport layer protocols are implemented in the package toast.transport. They extend the abstract class toast.transport.Comm (which registers the user when he joins a session), and implement the methods *sendUnicastUpdate* and *sendMulticastUpdate*. *sendUnicastUpdate* requires the identification of the destination host as a String. This identification is provided by the session server. The identification used is a String representation of the IP address, followed by the character ':' and the port number. This is network-layer dependant (we assume that the underlying network protocol is IP), but could easily be adapted to any other protocol by modifying the String sent by the session server. *sendLocalUpdate* is implemented by protocols that voluntarily delay the local display of changes in the application. When such a protocol is used, the update message is sent by the application to the transport layer, and then back to the application when processed, instead of being processed directly by the application.

Instantiating the class Comm when one transport protocol is loaded also creates one instance of the class toast.util.NetworkLayer. This instance, called socket, provides an interface to the network layer and simulates the network impairments. The send method simply sends the byte array given as argument (the format of this byte array is transport layer dependant, see chapter 3) in a UDP packet, the IP address and UDP port being given in the String representing the destination host. This means that actually the transport protocols are implemented "on top" of UDP, instead of being used directly on top of the network protocol, but this does not alter the theoretical and practical analysis that are made hereafter, since UDP merely adds port numbers and checksums to the transmitted packets and relays them transparently.

4.2.4 Simulating Network Impairments

Experiments have been conducted over a switched Ethernet LAN, which has negligible packet loss, delay and jitter. Network impairments need to be simulated. This has been done on the receiver side. The parameters *delay* (the average simulated delay), *jitter* (the maximum jitter), *minLossRate* and *maxLossRate* (the minimum and maximum probability that a given packet is lost) are read from a file.

Whenever a packet is received from the UDP socket, it is first delayed by $delay + j\ milliseconds$, with j a value chosen randomly between $-jitter$ and $+jitter$ with a uniform probability distribution. To do that, a thread loops indefinitely and listens to the UDP socket. When a packet is received, a delaying method is called (the parameter being the received packet). This method starts a new thread T_{packet} with the received packet and returns immediately, so that the initial thread can keep listening to the UDP socket. T_{packet} sleeps for $delay+$ j *milliseconds* and then adds the received packet to a FIFO queue. This FIFO queue represents the socket with the simulated delay added.

The packet loss is then simulated. When the transport layer calls the *receive* method of the network layer (this method has been given the same syntax as the java *DatagramSocket.receive* method), the FIFO queue is polled. If no packet is present, the method waits to be notified for a new packet. If one is present, a random number is chosen between 0 and 1. If this number is smaller than the loss probability, the packet is "lost" (i.e. discarded), and the polling of the FIFO queue begins again. This is done as long as packets are "lost". Finally, when a packet is not "lost", the *receive* method returns this packet.

The implemented loss model is actually a bit more complex than having a fixed loss probability. We have chosen $P(n)$, the probability that the n^{th} received packet is discarded by the simulation layer, to be given by:

$$\begin{aligned} P(n|n-1) &= P_{max} \\ P(n|\overline{n-1}) &= P_{min} \\ \therefore P(n) &= P_{max} * P(n-1) + P_{min} * P(\overline{n-1}) \end{aligned}$$

where $P(n|n-1)$ is the probability that packet number n is lost given that packet number n-1 has been lost, and $P(n|\overline{n-1})$ is the probability that packet number n is lost given that packet number n-1 has been received. This can be used to simulate loss bursts (a packet is more likely to be lost if the preceding packet was lost as well) that happen on the Internet.

4.2.5 Session Server

The framework also implements a session server, which provides each newcomer with a set of Strings identifying users already member of the session, and which sends to session users a String identifying any newcomer. The session server is a simple server accepting TCP connections by listening on a predetermined port and maintaing a table of users currently connected. It is a standalone program, independent from the application (it can be launched on a machine that does not run any related application and therefore does not run the TOAST framework).

To join a session, the toast.transport.Comm class opens a TCP connection towards the session server (the class has to be given the IP address of the session server and the port on which it accepts those TCP connections). Once

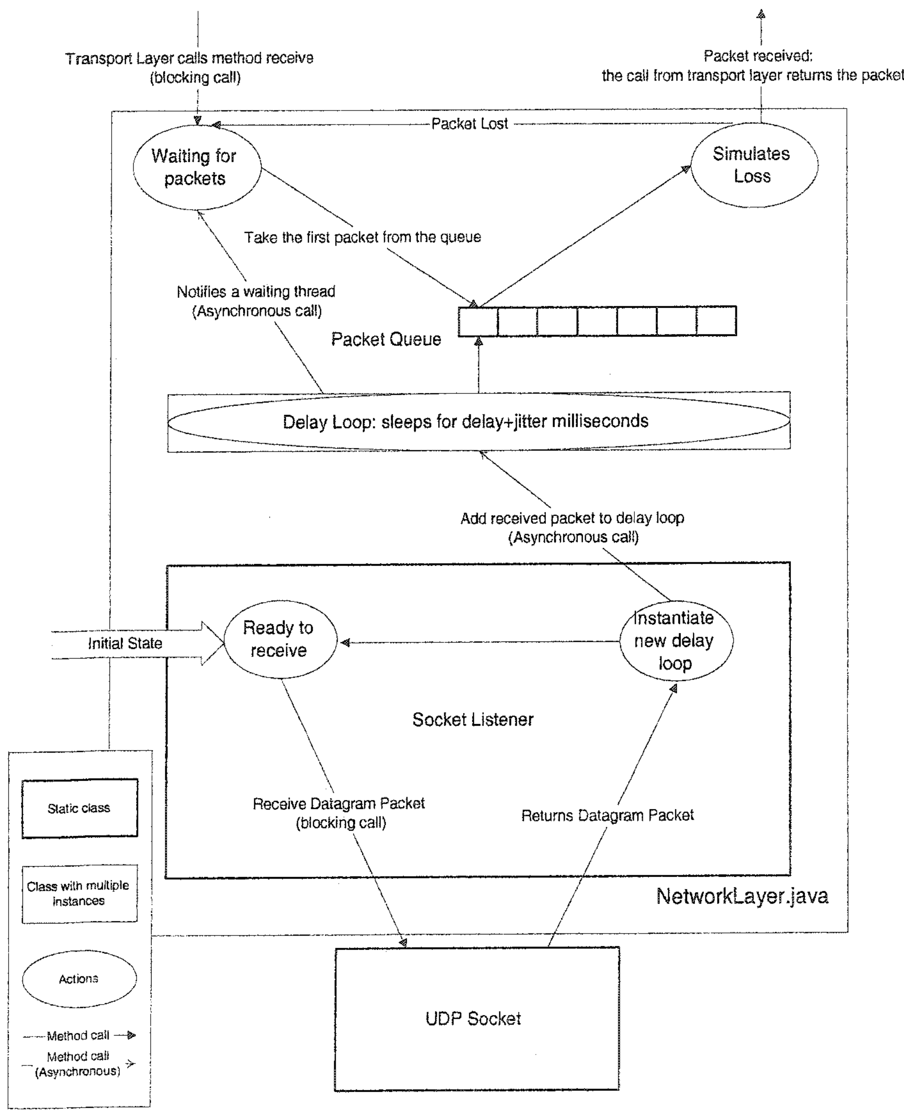

Figure 4.2: Network Impairments Simulation.

the connection is accepted by the server, the Comm class sends its own identity String, and waits indefinitely for identity Strings sent by the server. Whenever such a String is received, it is added to the local hosts table, the update messages being sent to all the members of the host table.[2] On the server side, the ID String read on the newly accepted connection is added to the session server table. This addition is sent to all current users, and the table (minus the new addition) is sent to the new user.

Connections have to be maintained throughout the whole session. A connection teardown (as a result, for example, of the application that established this socket exiting) will result in an exception in the session server, which will be interpreted as a user leaving the session. Subsequently, the users list is updated and all other users are sent the ID String corresponding to the lost connection with the instruction to remove it from their local list.

4.2.6 NTP Client

Finally, in order to have a synchronized time for all the participants in a given session, the framework also implements a NTP client. When the framework is instantiated, a NTP server is queried, and the difference between the local time and the NTP time, in milliseconds, is stored in a global constant. The common, synchronized time, is given by this constant added to the value returned by *System.currentTimeMillis()*.

4.3 Implementation Details

This section contains details about some issues encountered during the development of this framework, that might be of interest to developers of Java applications. This section can be skipped with no loss of continuity.

4.3.1 Java Native Interface

The main application being written in Java, the Java Native Interface (JNI) has been used to communicate with the C++ program using GHOST. The JNI is a standard tool provided by Sun that allows "native" methods compiled in

[2]If the network layer implements multicast, the exact ID of every host is irrelevant, since the network layer itself will forward the packets to every host taking part in the session. However, the session server is still needed to know how many hosts are currently taking part in the session, and therefore how many acknowledgments to wait.

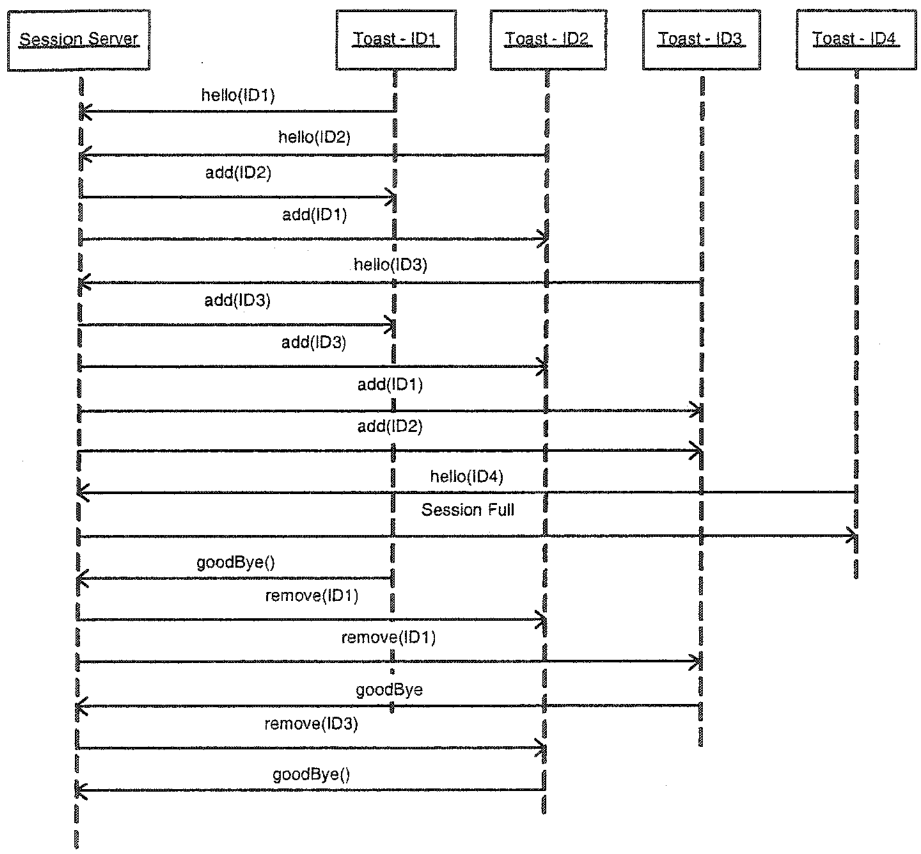

Figure 4.3: Communications with the Session Server, with four clients and MaxPopulation=3.

Java	C++
Package "pkg", class "classname"	Source code of nativeLib.dll
static{ System.loadLibrary("nativeLib") }	*#include <jni.h>*
public native type function (type1,type2);	*JNIEXPORT jtype JNICALL Java_pkg_classname_function (JNIEnv*, jobject, jtype1, jtype2)*

Figure 4.4: Conventions of the Java Native Interface.

a machine-dependent format (here, the Microsoft Dynamic Loadable Library (**DLL**)) with method names following a precise convention to be called from the Java program [40]. Figure 4.4 gives an overview of these conventions.

Methods on the native side have access to JNIEnv*, a pointer that accesses some JVM's methods (needed for memory management, among other things) and can even instantiate a virtual machine if needed, and jobject is an equivalent of "this" for the native method. Java primitive types are translated into JNI-defined types. For example, a "double" argument in java becomes a "jdouble" on the native side.

4.3.2 Navigating in a Simple Universe

The description of a Java3D universe made in chapter 2 is actually the description of a *simple universe*, used in our implementation. The main difference between a simple universe and a general Java3D universe is that the latter provides a complex viewing architecture, separated from the scene graph, whereas the former only provides a simple viewing platform. This proved to be enough for the box carrying application presented here, but the transformation that is applied to the viewing platform must be chosen carefully. In the remainder of this section $Tx_{Initial}$ is the initial transformation matrix associated with viewing platform. $Tx_{Applied}$ is the transformation applied to this matrix, and Tx_{Final} is the final transformation matrix applied to the viewing platform, which needs to be computed.

If $Tx_{Applied}$ is a translation, $Tx_{Final} = Tx_{Initial} * Tx_{Applied} = Tx_{Applied} * Tx_{Initial}$ makes the user moves with respect to the virtual world, keeping the same orientation.

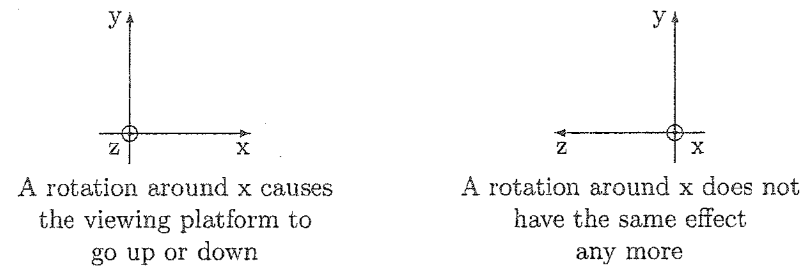

Figure 4.5: Two Viewpoints for the User.

If we want the user to have the illusion that the world is rotating around him, but that he remains at a fixed point, $Tx_{Applied}$ is a rotation matrix, and $Tx_{Final} = Tx_{Initial} * Tx_{Applied}$.

If we want the user to have the illusion that he is rotating around a fixed point, $Tx_{Final} = Tx_{Applied} * Tx_{Initial}$ has to be chosen. A problem arises in the definition of $Tx_{Applied}$: the rotation has to be given in the user's referential, not in the virtual world fixed referential. For example, if the user is to "go up", in the first position shown in figure 4.5, $Tx_{Applied}$ is a rotation around the x axis, with a negative angle. But if later the same user is to "go up" being in the second position, a rotation around the x axis will not have the desired effect.

This is why $Tx_{Applied}$ is defined as a rotation around one of the fixed-coordinates axis, but instead of applying $Tx_{Final} = Tx_{Applied} * Tx_{Initial}$, a slightly more complex transformation is applied: $Tx_{Final} = R_{Initial} * Tx_{Applied} * R^{-1}_{Initial} * Tx_{Initial}$, with $R_{Initial}$ the rotation component of $Tx_{Initial}$. As a general rule, $R * Tx * R^{-1}$ corresponds to a referential change for any transformation Tx and any rotation R between the old referential and the new one.

4.3.3 Simple Ownership Management

The tracheostomy simulation that we developped requires ownership management so that only one user at a time can grab a given tool. The following algorithm has been used to request a tool with index i:

- indexlocked $\leftarrow$ i
- for every remote host: send(request for i)
- for every reply received: if reply is true, indexlocked $\leftarrow$ 0, return true
- return false

Therefore, if at least one remote host replies "true", the method returns true and the application cannot take ownership of the object. If all the remote hosts reply "false", the tool is currently free and the application grabs it.

On the reply side, when a request for the index i is received, the application does:

- if indexlocked equals i send true
- else send false

The variable *indexlocked* is used to ensure that if two applications request the same tool at the same time, at most one gets the ownership of the tool.

Another way to implement ownership management would be to modify the session server and add to each participant the list of owned shared objects. In this case, requests for ownership would have to be sent to the session server, which could decide whether the requested ownership can be granted or not, and send its reply.

Chapter 5

Two VE Applications

In order to test the framework and the protocols presented in preceding chapters, two applications using a haptic interface and network communications have been developed.

5.1 Tracheostomy Training

5.1.1 Aim

The goal is to simulate a surgical act commonly performed in emergency medicine, tracheostomy. In the scenario presented here, two surgeons, or trainees, have to share their tools (a scalpel, two surgical hooks and a piece of gauze) to cut the skin on a virtual patient's throat, spread it open and cut inside the

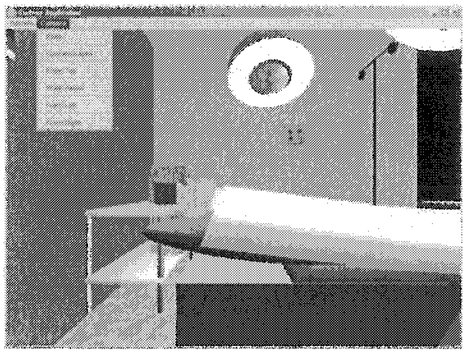

Figure 5.1: The Operating Room.

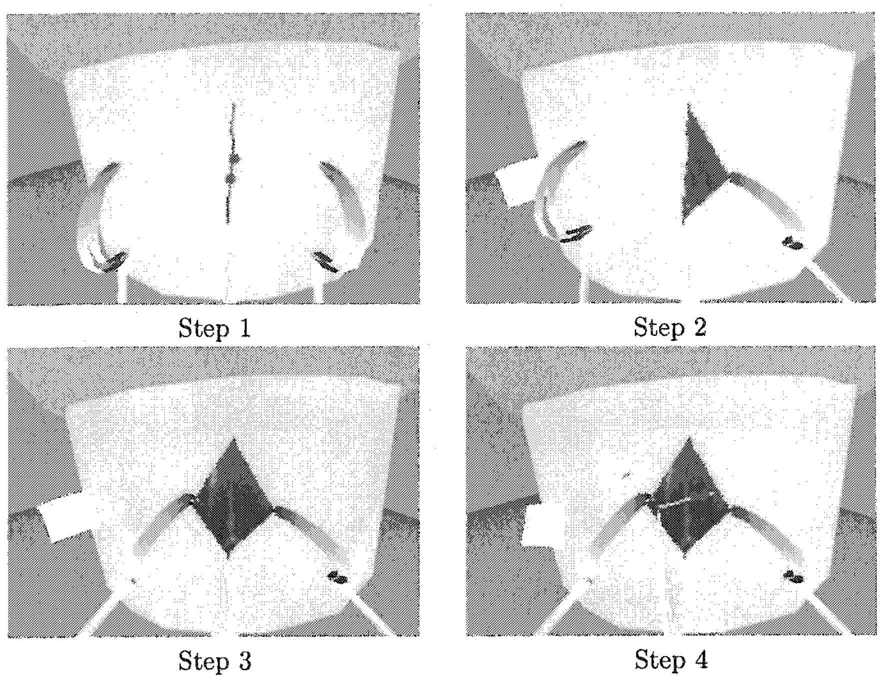

Figure 5.2: Tracheostomy: the Operation Step by Step.

underlying muscle layer. This application is based on patient and tools models previously developed at DISCOVER Lab.

5.1.2 Application Scenario

The following scenario has been followed during each simulation session, involving two users:

- User A grabs the scalpel and performs a vertical cut (Step 1 in figure 5.2);
- User B grabs the gauze to remove spilling blood;
- User A takes successively both hooks to pull the skin (Steps 2 and 3);
- User B performs a horizontal cut on the muscle (Step 4);
- User A cleans the blood coming from the second cut.

The users were instructed to perform those operations as quickly as possible, and total execution time has been measured for each trial.

5.1.3 Architecture

This application uses the playback architecture presented in [10], which among others, loads VRML descriptions in Java3D and handles avatar behaviors (so that navigation through the operation room is possible). The main class of the application extends *COSMOSApplication*, an abstract class from the playback architecture. After the playback architecture is initialized, a new Java3D universe is instantiated, with the the following objects:

- The description of the room (a branch graph generated from a VRML description);
- A new avatar, with a behavior listener that allows him to move in the room. The universe's viewing point is attached to this avatar (and therefore, the avatar itself cannot be seen by the application user);
- Lights;
- Skin (a branch group whose aspect will be modified according to the users' actions);
- A muscle layer, a visual representation of the layer under the skin;

Then the Graphical User Interface (**GUI**), that displays the Java3D canvas, is started, and the PHANToM is initialized.[1] The *tool manager* is also started.

The application is *distributed*. Therefore, each user has to maintain locally the state of the patient's throat, according to the user's input and to the updates received via the network. There is no latecomer support, since a user who joins the session late has no means of knowing in what state the patient's throat is (whether or not incisions have been made, at which position...).

5.1.4 The Tool Manager

The tool manager is responsible for:

- Loading the two surgical hooks, the scalpel and the gauze in the application. Those devices are described in VRML;
- Creating a thin red cylinder that displays the position of the PHANToM in the graphic scene when no tool is selected;

[1] If a playback session, and not a simulation session, is initiated, the haptic interface is not called. This playback session will be described shortly at the end of the section.

- Reading the translation and rotation coordinates from the haptic interface and applying them to the tool currently held by the user, or to the red cylinder;
- Sending update messages corresponding to those movements;
- Applying the update messages received from other hosts to the other tools;
- Managing ownership: the user may take control of a tool only if no other user is currently using it.

Since update messages are handled by the tool manager, this class extends *toast.Toast*. A separated thread processes the update messages. They are received either from Toast, or from the haptic interface. Every time an update message is generated, this thread is notified. It processes all the update messages that arrived since last run (there may be more than one, since update messages can arrive during the execution period of this thread). For every update message, the tool given by the index is updated in position and orientation, and the skin layer is notified. For each tool, a corresponding method in the skin layer is called with the position of the tool as parameter. The thread then waits for a new notification to arrive.[2]

This class is synchronized with the rendering part of Java3D. The haptic interface is not polled at fixed intervals, but once per execution of the rendering thread. A callback has been added in the *preRender* method of Canvas3D, so that just before the canvas is rendered, the haptic interface is polled for the current tool's position and orientation, and the graphic scene is updated accordingly.

5.1.5 The Haptic Interface

The Haptic Interface is an interface (via the JNI and a C++ DLL) to the GHOST SDK described in chapter 4. It loads a VRML description of the throat in the haptic interface (this VRML description is felt by the user), and provides a method that returns the PHANToM position and orientation. When queried, this raw data has to be corrected: $c \leftarrow g_c * (c - o_c)$ for $c = x,\ y,\ or\ z$. o_c is an offset and g_c a gain. They have been determined experimentally to synchronize

[2] The thread does not wait for an indefinite amount of time. After 200 milliseconds, it wakes up, and forces the renderer to run again. This has been done to prevent deadlocks when no update messages are received from toast, since the renderer waits for scene modifications to run and scene modifications are done only after a callback from the renderer.

the device with the graphical representation of the scene. An offset is subtracted from each angle as well. Those operations are performed in the tool manager class.

5.1.6 The Skin Layer

The skin layer is a branch group containing n^2 square skin elements. In the implementation presented here, $n = 80$ has been chosen. Each skin element can have three different appearances: the normal skin appearance, a transparent appearance (used when this element is pulled by a hook, to reveal the muscle layer underneath), and a yellow appearance, used to symbolize the cut on the muscle layer (this way, changes are made to the skin layer only, and the muscle layer is a simple picture). Moreover, at each skin element a circular animated shape may be dynamically added to simulate blood drops.

The class skin has one method for each surgical tool, which is called by toolManager. The method cut changes the appearances of all the pixels on the segment between the preceding point that has been cut and the current point. Those pixels are made transparent. If a point of this segment is situated over a blood vessel, a blood drop is added. If the cut happens on already transparent points, they are given the yellow appearance, to symbolize a cut on the muscle layer.

The pull method defines a triangle between the current point and the two ends of the cut. All the skin elements within this triangle are made transparent.

The clean method removes all the blood drops in the gauze area.

5.1.7 Playback

Since all the inbound and outbound update messages are logged by the TOAST framework, it is possible to play back the movements of the tools and their action on the skin during a given session. If the playback option is chosen at the start of the application, the tool manager class and haptic interface are not instantiated. Instead, the incoming and outgoing log files for a given host in a given session are parsed, and all the update messages are stored in a vector with their timestamps (that are also provided by the logs). Once the logs are parsed, a timer is started, at a frequency $frameRate$ chosen by the user. Every $\frac{1}{frameRate}$ second, the timer checks the vector, retrieves all the update messages belonging to the considered time slot, and applies the corresponding transformations to the tools and the skin (independently from the original direction of the displayed

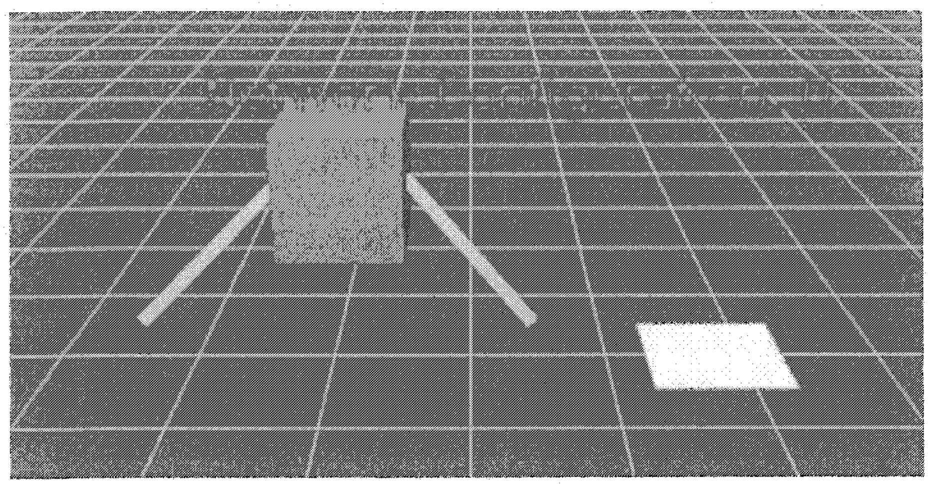

Figure 5.3: The Box Carrying Application.

update message). This way, tools are moving and the skin is cut, pulled and cleaned the same way it had been during the simulation session.

Finally, there is the option to capture every frame by the time it is displayed, at regular intervals. This generates JPEG-encoded pictures, which put together make a video file showing the simulation session. The advantage of this approach is that the video can later be played by any video player, and does not require to load the whole application, nor the installation of the JVM and the Java3D extension.

5.2 Box Carrying

5.2.1 Aim

This application is not intended to be realistic, but to give better results than the tracheostomy application about collaborative work. Two users have to carry, in the presence of gravity, a virtual box sideways, then towards them, using the PHANToM interface. The box may be grabbed by two handles, situated on two opposite sides of the box. Aside from the box (the only shared object in this application), there are two hooks (each hook being manipulated by one user), and the ground.

The collaboration is imposed by the gravity: if both users do not tightly synchronize their actions, the box starts falling, and the grip on the handles may be lost.

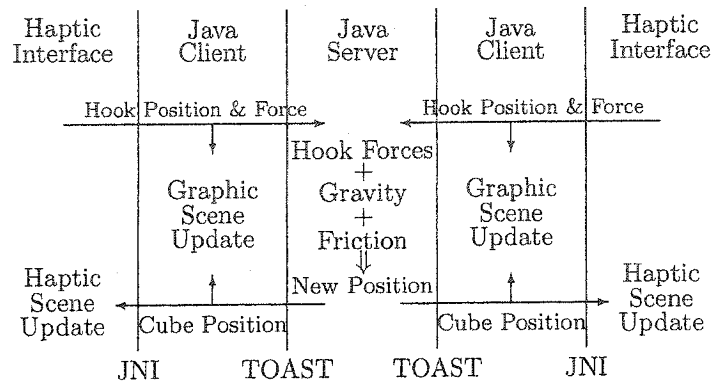

Figure 5.4: Message Passing in the Client-Server Architecture.

5.2.2 Client-Server Architecture

The client server implementation ensures that a single host (the server) receives and processes the updates for the shared object. Two update messages received concurrently (within the same period of 10 milliseconds) from each user are processed at the same time and yield a combined effect. Doing so in a distributed way (i.e. having each user computing the cube position) cannot result in a consistent result, and makes the whole application unusable since reaction forces created when the hook touches the box depend on the exact position of the box.

Each user (client) handles a hook. Its position follows the position of the PHANToM. The position of the hook and the force applied by the PHANToM are sent to the server. The server applies those forces to the shared object and computes the resulting position of the box.[3] If the position has changed, an update is sent to the clients.

The client has two representations of the world: a graphic one (through Java3D) and a haptic one (through GHOST). Each representation contains a cube representing the box, that is moved accordingly to the update messages received from the server. This exchange is shown in figure 5.4.[4]

[3]Torque has not been taken into account, so the box cannot rotate on itself under the action of the hooks.

[4]Clients also send the position of the local hook to each other, which is not shown on the figure.

5.2.3 Server Implementation

The server keeps as a local variable the values of the hooks position and applied force. Any received update message modifies those values. Hence, if several update messages arrive for the same hook before the value is taken into account, only the latest value is considered.

A timer causes the main method of the server to run every 10 milliseconds. During each run, the positions and forces applied by the hooks are read and reset to zero. Doing so ensures that a given force is never applied more than once. Those forces apply only if the hook is touching a handle (therefore the need to check the position). Gravity is added to the sum of those forces, as well as friction force if the box touches the ground. This force is then multiplied by a constant value and added to the current position of the cube. If the position of the cube has changed (i.e. if the sum of the forces is not zero), an update message with the new position of the cube is sent to the two clients.

The way forces are applied on the cube (multiplication by a constant value and addition of the result to the cube's position) does not yield a realistic force effect (to achieve that, the force should be integrated twice over time), but the aim of the application is not to be as realistic as possible, and this simple approximation is enough for our purpose.

Since no human interaction happens at the server, buffering is not required. This is why during the tests, when the clients used smoothed SCTP, the server relied on SCTP for transport. Since they share the same packet format, the two protocols coud collaborate, and the enforced delay was subsequently decreased. The maximum delay in this case is the RTT between the client and the server, instead of twice this value if the server had been running smoothed SCTP as well.

5.2.4 Client Implementation

The client creates the ground, a colored cube and cylinder representing the hooks and add them to a new Java3D universe. The viewing point can be moved through keyboard inputs. To the colored cube, two spheres representing the handles are added. They belong to the same transform group as the cube, so that they move along with it. A haptic scene is also instantiated. The position of the user's hook is read from the haptic interface and the graphic representation of the hook is modified accordingly. If the force applied to the cube is not zero, an update message containing the position of the hook and the applied force is sent. When an update message is received, the graphic display of the corresponding object (either the box or the other hook) is modified accordingly. If the position of the cube changes, the haptic interface updates the haptic scene as well.

Haptic Interface

The haptic interface, when instantiated, creates a cube and two force fields that draw the tip of the PHANToM towards the handles, and make it stick to those handles when they have been reached. The method *getLastPosition* returns the position, rotation angles and reaction force applied to the PHANToM by the GHOST API. The force that the PHANToM applies to the box is the opposite of the reaction force applied to it.

The two force fields are magnetic fields, that attract the PHANToM towards them. The force is given by $\vec{F} = \frac{1}{K+d}\vec{u}$ where K is a constant (added to avoid dangerously big forces when d is small), d is the distance between the PHANToM tip and the position of the handle and $\vec{u}$ is a unity vector giving the direction of the vector going from the PHANToM tip to the handle. If we keep applying this force when the PHANToM is very close to the handle, instability occurs and the device vibrates: when the tip of the PHANToM is close to the target, a big force is applied. This big force causes the tip to move closer to the target, then to pass on the other side. The applied force is then reversed in its direction, and the same phenomenon takes place. When the tip of the PHANToM keeps moving around the centre of the force, a vibration is felt.

To avoid that, the magnetic force is replaced by another force when the PHANToM is close to the target. To provide stability, damping is introduced. More precisely, a spring-damper model (as shown in figure 5.5) is used: the force applied is proportional to the squared distance between the tip and the target point (this is similar to the force felt when stretching a spring), but reduced by an amount proportional to the speed of the tip (fluid damping).

To this force, the reaction force, $\vec{F_R}$, computed by the GHOST API, is automatically added in case the tip of the PHANToM touches the box. Finally,

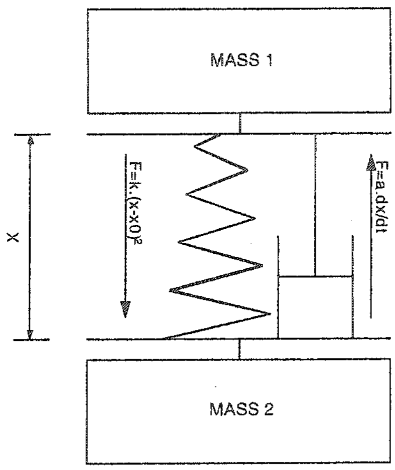

Figure 5.5: The Spring-Damper Model.

the total force felt by the user is:

$$\vec{F}_{total} = \begin{cases} \vec{F}_R & \text{if } d > d_{bound} \\ \vec{F}_R + \frac{K}{1+d}\vec{u} & \text{if } d > d_{thres} \text{ and } d < d_{bound} \\ \vec{F}_R + (K_1 d^2 - K_2 \dot{d})\vec{u} & \text{if } d < d_{thres} \text{ and } K_1 d^2 - K_2 \dot{d} > 0 \\ \vec{F}_R & \text{if } d < d_{thres} \text{ and } K_1 d^2 - K_2 \dot{d} < 0 \end{cases}$$

where d_{bound} is one third of the size of the cube's edges, d_{thres} has been arbitrarily set to 5mm,[5] and the constants K, K_1 and K_2 have been set so that the module of the force is continuous when $d = d_{thres}$. $\dot{d}$ is the speed of the tip of the PHANToM.

[5]An implicit assumption is that the cube's edges measure more than 3 times 5mm.

Chapter 6

Data Retrieval and Test Results

6.1 Procedure

For each application, and each protocol under trial, 5 sets of measurements have been made. In each set, the task has been accomplished several times, with a change in the network conditions between each execution. The two users are the same for all the trials. They are not skilled in surgery, and a preliminary training gave them confidence in the use of the PHANToM and in the applications themselves.

Simulations without human presence in the control loop would greatly increase test possibilities since many network configurations could be tested without the need for users to be present (which require them to synchronize their schedules, to ensure that the hardware is available...). However, this would require a precise model of human actions in this application, which is unfortunately a very complex task. To the best of the author's knowledge, no such model is available today.

It is acknowledged that a general proof of viability of the protocols is not given by the results presented here, but the relative effectiveness of the protocols for collaborative tasks can be deduced from the results of the box carrying application.

6.2 Total Execution Time

The total execution time is given in the tracheostomy application by the difference between the completion date (when the gauze finishes cleaning the blood spilled after the second cut) and the beginning date (taken as the date at which the scalpel or the gauze starts moving, whichever happens last).[1] In the box carrying application, the completion date is the time by which the box reaches its goal, and the beginning date is the time at which both users have taken hold of the box. All the times in this section are given in seconds.

Network conditions	SCTP	Smoothed SCTP	Light TCP
No impairment	26.0	29.9	29.3
20% loss	36.2	31.0	30.9
50% loss	37.6	34.0	29.6
10 to 50% loss	23.8	26.1	32.3
50ms delay, 50ms jitter	30.4	32.3	35.7
100ms delay	30.0	37.1	31.2

Table 6.1: Total Execution Time, in seconds, for the Tracheostomy Application.

Network conditions	SCTP	Smoothed SCTP	Light TCP
No impairment	17.6	10.6	127.1
20% loss	29.4	16.6	210.6
30% loss	62.9	24.1	N/A*
10 to 50% loss	39.9	21.1	N/A*
50ms delay	14.9	18	63.2
50ms delay, 10ms jitter	39.5	16	N/A*

* N/A means that the box could not be carried to the goal

Table 6.2: Total Execution Time, in seconds, for the Box Carrying Application.

Results for the tracheostomy simulation, shown in table 6.1 tend to exhibit a relatively constant performance when using Light TCP, when there is no jitter (around 30 seconds). Paradoxically, SCTP seems to perform better than smoothed SCTP in presence of elevated delay and/or jitter, and worse when packet loss is present, when the opposite was awaited. However, this series of

[1]The first user, who holds the scalpel at the beginning, had been instructed not to move before he saw that the other user had grabbed the gauze.

measurement do not appear to be very significant, due to a big standard deviation in the measures performed for the same protocol and under the same conditions. It is believed that this application is too complex to allow reproducible performance within the same time in the same conditions. Moreover, the tracheostomy application does not appear to be sensitive to collaboration issues. Users have to share their tools, but they are never required to use any given tool at the same time. Therefore, except at the times at which the users have to change their tools, they remain largely independent from the other user, and consequently from the network impairments.

In the box carrying application (see table 6.2), the use of Light TCP totally disables the collaboration. In the case when the goal can be reached, the execution time is one order of magnitude higher than the time with the two other protocols. As expected, smoothed SCTP helps the users carrying the task more efficiently in the presence of delay. The same is also true in the presence of packet loss (which can be seen as an extreme form of jitter), and yields a relatively constant execution time. The price to pay is that in the presence of delay only (which can be a realistic assumption on leased ATM links), performances are not as good as those achieved with SCTP.

6.3 Number of Errors

For the box carrying application, collaboration errors are defined as the number of times the hook leaves the area where it can grab the box during one session (see table 6.3).

Network conditions	SCTP	Smoothed SCTP	Light TCP
No impairment	30.9	28.7	283
20% loss	54.2	28.7	537
30% loss	126.7	38.2	N/A*
10 to 50% loss	101.6	33.3	N/A*
50ms delay	33.0	19.2	161
50ms delay, 10ms jitter	87.8	20.8	N/A*

* N/A means that the box could not be carried to the goal

Table 6.3: Average Number of Collaboration Errors.

Most of the times, this error results from a user moving too fast (when pulling

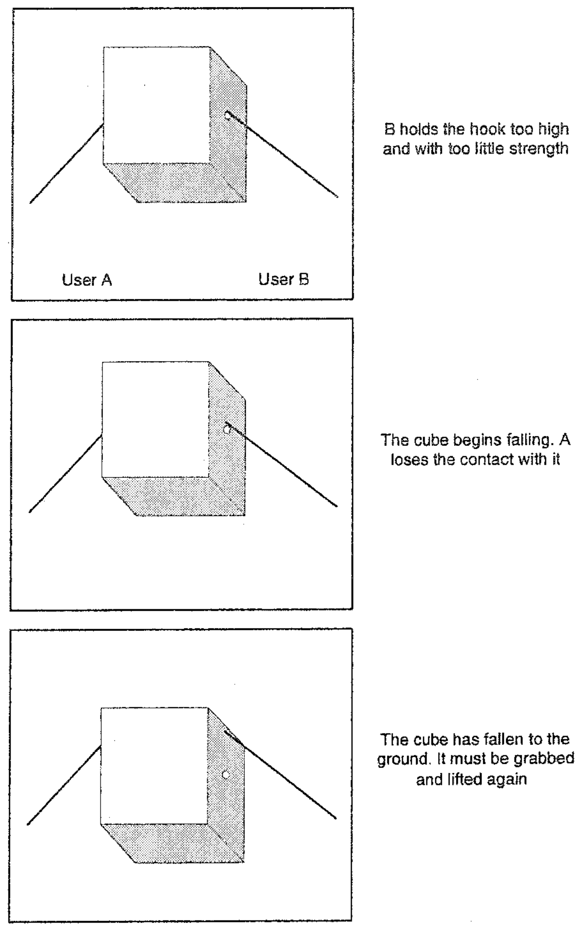

Figure 6.1: A Collaboration Problem Leading to a Failure.

the box) or too slowly (when pushing the box), or maintaining the position of the hook too high when the gravity makes the box going downwards.

If one of the users makes a collaboration error, and subsequently stops applying a force to the box, the other user, alone, is unable to counter gravity alone, so he makes a collaboration error as well, as illustrated in figure 6.1. The box, being subject only to the gravity, falls, and the two users have to grab it again and resume carrying it, which increases the execution time. Therefore, errors and execution time are dependent variables in this application, as shown by figure 6.2.

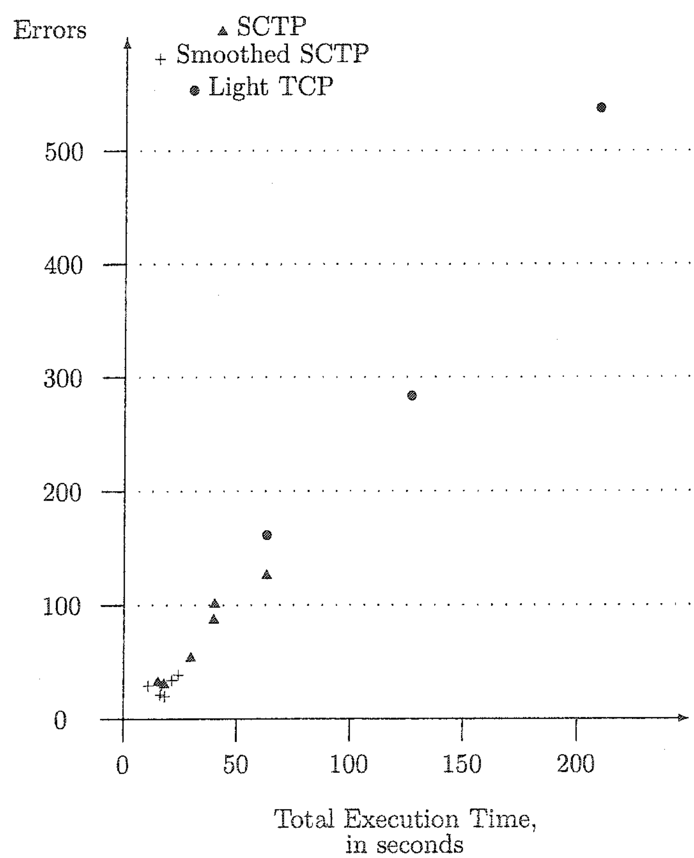

Figure 6.2: Dependency between Error Numbers and Execution Time.

Chapter 7

Conclusions and Future Work

7.1 Solved Issues

In this thesis, we have studied network impairments in CVEs. To solve the problems created by delay, jitter, and packet loss, we took several transport-layer protocols into consideration and evaluated their relative efficiency.

7.1.1 Queuing on the sender side

Despite the improvements that Light TCP brings as compared to the dominant TCP reliable transmission model, the former protocol is still too close to TCP to provide efficient collaboration. The results found suggest that no queuing should actually be done on the sender side as long as haptic update messages are involved. They have to be sent as soon as they are produced by the application.

7.1.2 Queuing on the receiver side

On the opposite, queuing update messages after they arrived at their destination is an interesting approach. Smoothing out jitter, at the price of an additional delay, significantly improves the way users can collaborate. A parallel can be drawn with techniques devised to carry human voice: delay and jitter are also important parameters, especially in a conversation between two persons. Transmitting haptic update messages is more challenging than transmitting voice since timely reliability is required. However, in both cases jitter and delay decrease or destroy the user's perception of the remote user, and in both cases jitter can be smoothed out at the expense of additional delay. Therefore, the combination of SCTP, to ensure timely reliability, and buffering on the receiver side, is an efficient solution.

7.1.3 Applications design

When designing a distributed application, one of the most fundamental issues that arises during the specification phase is whether to implement a peer-to-peer or a client-server application. Client-server implementations are less fit for scalability, and increase the end-to-end delay, because the client has to send an update to the server, which then relays it to another client, which takes twice as long as directly sending an update from a peer to another. However, client-server is more fit for collaborative applications since the server is able to maintain a coherent description of the virtual universe.

The importance of delay awareness has also been shown. This can be implemented either at the application level, with application-specific techniques like predictive overlay, or at the transport layer, with smoothed SCTP.

7.2 Related Issues

7.2.1 Preserving Bandwidth

As we have seen, queuing on the server side is not desirable. One of the original goal of this queuing was to reduce the bandwidth needed by the application. There are several other ways to reach this goal, but none of them is application-transparent:

- reducing the size of the update messages (e.g. by using single-precision floating point numbers instead of double-precision ones);
- lowering the rate at which update messages are generated (there is however a lower limit to this rate under which collaboration is not possible any more);
- bundling update messages to decrease protocols overhead. This is only possible at the application level, by bundling update messages that are

produced simultaneously for different haptic objects, but that share a semantic link;

7.2.2 QoS Management

QoS can be offered through:

- Admission control;
- Shaping;
- Policing;
- Routing;
- Scheduling;
- Buffer Management;
- Traffic Monitoring.

Several of these components involve queuing or buffering on the sender side. From what has been said above, it seems likely that using a network that provides QoS may actually be harmful to collaboration, if the haptic update messages are not given the highest priority.

7.2.3 Complex Haptic Devices

This thesis took a relatively simple haptic device into account. An issue that has not been solved here is to know whether SCTP and smoothed SCTP would support more complex haptic devices efficiently. Data gloves equipped with force feedback for each finger, or even each joint, have up to 26 DOF: the hand itself has six DOF, and each finger has four more. This is significantly more than the six DOF that have been considered for the PHANToM. Other devices modelling the state of the skin are foreseeable. They would bring the requirements one order of magnitude higher.

There is no theoretical limit to the number of DOF used in the update messages carried by the transport protocols. However, all the network protocols used today (and the underlying data link protocols as well) impose a maximum packet size. If transport-layer packets are bigger than the highest allowed payload, they are either rejected or fragmented. Not only does fragmentation increase packet processing time, it also adds potentially large amounts of jitter in best effort networks, since packets are delivered to the transport layer on the receiver side only after all the fragments have been received, and there is no guarantee that all the fragments are carried along the same path. Therefore, fragmentation of large update messages can increase the perceived network impairments.

7.3 Future Work

SCTP, smoothed SCTP, and TOAST specifications rely on several parameters:

- Network delay and jitter;
- Key update frequencies;
- Retransmission timeout;
- Enforced delay;
- Bucket length.

All those parameters were constant values in our experiments. The implementation should allow their dynamic modification, and parameters adaptation to the network conditions.

More trials need to be done to determine what is the maximum enforced delay that users can tolerate, and whether dynamically modifying this enforced delay (on a larger time scale than transmission time, to avoid creating a new source of jitter) would be acceptable or not.

The performance of the protocols themselves can be increased by implementing them in a way that is much closer to the available hardware. This is possible, for example, by implementing the specifications given in this thesis in C, and integrating the protocols to the Linux Kernel, so that the operating system directly passes packets received from the network interface card to our transport protocols.

Finally, the logging provided by TOAST has been found to be insufficient to easily provide readable and conclusive results. The extensive logging of the passed update messages is not enough to ensure that raw data will later be processable. One solution would be to log, not only the update messages, but also application-dependant messages in a separate log (this would yield a fifth file, beside the update messages received and sent logs, and the network layer traffic logs). This log would contain messages like "failure" or "scalpel starts cutting" to which a timestamp is appended. The number of messages that the application can log should be limited and specified, and the log parser written according to those specifications.

Bibliography

[1] M. C. Çavuşoğlu, F. Tendick, M. Cohn, and S. S. Sastry, "A laparoscopic telesurgical workstation," in *IEEE Transactions on robotics and automation*, vol. 15, pp. 728–739, IEEE, August 1999.

[2] F. Arai, M. Tanimoto, T. Fukuda, K. Shimojima, H. Matsuura, and M. Negoro, "Multimedia tele-surgery using high speed optical fiber network and its applications to intravascular neurosurgery–system configuration and computer networked implementation," in *Proceedings of the IEEE International Conference on Robotics and Automation*, vol. 1, pp. 878–883, 1996.

[3] MPB Communications, Inc., "Live simulated surgery: a "touching" success." `http://www.mpbc.ca/mpbc_2004/main_pages/news/2002/6dof.html`, December 2002.

[4] D. Bielser, *A Framework for Open Surgery Simulation.* PhD thesis, Swiss Federal Institute of Technology, ETH, Zurich, 2003.

[5] SensAble Technologies, Inc., "SensAble Technologies announces FreeForm® Concept^{TM} version 1.0" `http://www.sensable.com/newsevents/pressreleases/`, November 7th 2003.

[6] S. Singhal and M. Zyda, *Networked Virtual Environments.* Addison Wesley, Reading MA, 1999.

[7] National Capital Institute of Telecommunications (NCIT), "Harmonie: Haptic, augmented reality multimedia for networked interactive environments." `http://www.discover.uottawa.ca/research/HARMONIE.html`, 2002.

[8] G. Bell, A. Parisi, and M. Pesce, "The virtual reality modeling language, version 1.0 specification," tech. rep., Web3D consortium, 1995.

[9] Sun Microsystems, Inc., "Java3D api." `http://java.sun.com/products/java-media/3D/`.

[10] M. Hosseini, "An architecture for recording and playback of MPEG4-based collaborative virtual environments," Master's thesis, School for Information Technology and Engineering, University of Ottawa, 1999.

[11] A. Bloomfield, Y. Deng, J. Wampler, P. Rondot, D. Harth, M. McManus, and N. Badler, "A taxonomy and comparison of haptic actions for disassembly tasks," in *Proceedings of the IEEE Virtual Reality*, 2003.

[12] SenSable Technologies, Inc., "Ghost SDK API reference." `http://www.sensable.com/support/phantom_ghost/datafiles/GHOSTAPIReferenceManual.pdf`.

[13] W. Broll, "Interacting in distributed collaborative virtual environments," in *Proceedings of the IEEE Virtual Reality Annual Symposium*, pp. 148–155, 1995.

[14] K. Brandenburg, "Low bitrate audio coding-state-of-the-art, challenges and future directions," in *Proceedings of the 5th International Conference on Signal Processing, WCCC-ICSP.*, 2000.

[15] R. Steinmetz and K. Nahrstedt, *Multimedia: Computing, Communications and Applications (IMSC Press Multimedia Series)*. Prentice Hall PTR, 2002.

[16] K. S. Park and R. V. Kenyon, "Effects of network characteristics on human performance in a collaborative virtual environment," in *Proceedings of IEEE Virtual Reality*, pp. 104–111, March 1999.

[17] M. Meehan, S. Razzaque, M. C. Whitton, and J. Frederik P. Brooks, "Effect of latency on presence in stressful virtual environments," in *Proceeding of the IEEE Virtual Reality*, 2003.

[18] D. Wang, K. Tuer, M. Rossi, L. Ni, and J. Shu, "The effect of time delays on tele-haptics," in *2nd IEEE International Workshop on Haptic, Audio and Visual Environments and their Applications - HAVE*, pp. 7–12, September 2003.

[19] T. B. Sheridan, "Space teleoperation through time delay: Review and prognosis," *IEEE Trans. on Robotics and Automation*, vol. 9, pp. 592–606, October 1993.

[20] L. Ni, *Position-error-based Force-reflecting Teleoperation*. PhD thesis, University of Waterloo, 2002.

[21] S. Shirmohammadi, *Synchronous Collaboration in Virtual Environments: Architecture, Design and Implementation*. PhD thesis, School for Information technology and Engineering, University of Ottawa, 2000.

[22] M.Eraslan, N. D. Georganas, J. R. Gallardo, and D. Makrakis, "A scalable network architecture with dynamic qos over ipv6 for distributed virtual environments," in *Proceedings of the 8th IEEE Symposium on Computers and Communications*, June 2003.

[23] N. Y. Chong, T. Kotoku, K. Ohba, and K. Tanie, "Virtual repulsive force field guided coordination for multi-telerobot collaboration," in *Proceedings of the IEEE International Conference on Robotics and Automation*, May 2001.

[24] N. Y. Chong, S. Kawabata, K. Ohba, T. Kotoku, K. Komoriya, K. Takase, and K. Tanie, "Multioperator teleoperation of multirobot systems with time delay: Part I-aids for collision-free control," *Presence*, vol. 11, pp. 277–291, June 2002.

[25] B. Watson, N. Walker, P. Woytiuk, and W. Ribarsky, "Maintaing usability during 3D placement despite delay," in *Proceedings of the IEEE Virtual Reality*, 2003.

[26] L. Gautier, C. Diot, and J. Kurose, "End-to-end transmission control mechanisms for multiparty interactive applications on the internet," in *Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, vol. 3, pp. 1470–1479, March 1999.

[27] D. Mills, "Simple network time protocol (SNTP) version 4 for IPv4, IPv6 and OSI," RFC 2030, IETF, 1996.

[28] International Organization for Standardization, "Information technology - Open Systems Interconnection - basic reference model: The basic model," Norm ISO/IEC 7498-1:1994, 1994.

[29] J. Postel, "Transmission control protocol," RFC 793, IETF, 1981.

[30] J. Postel, "User datagram protocol," RFC 768, IETF, 1980.

[31] D. R. Cheriton and D. Skeen, "Understanding the limitations of causally and totally ordered communication," in *ACM SIGOPS Operating Systems Review , Proceedings of the fourteenth ACM symposium on Operating systems principles*, vol. 27, December 1993.

[32] G. D. Kessler and L. F. Hodges, "A network communication protocol for distributed virtual environment systems," in *Proceedings of the IEEE Virtual Reality Annual International Symposium*, pp. 214–221, 1996.

[33] S. Shirmohammadi and N. D. Georganas, "An end-to-end communication architecture for collaborative virtual environments," *Computer Networks Journal*, vol. 35, pp. 351–367, February 2001.

[34] M. Mauve, "RTP/I – toward a common application level protocol for distributed interactive media," *IEEE Transactions on Multimedia*, vol. 3, pp. 152–161, March 2001.

[35] M. R. Macedonia and M. J. Zyda, "A taxonomy for networked virtual environments," *IEEE Multimedia*, vol. 4, pp. 48–56, Jan-Mar 1997.

[36] M. Pullen, M. Myjak, and C. Bouwens, "Limitations of internet protocol suite for distributed simulation in the large multicast environment," RFC 2502, IETF, 1999.

[37] J. M. Pullen, "Reliable multicast network transport for distributed virtual simulation," in *3rd IEEE International Workshop on Distributed Interactive Simulation and Real-Time Applications.*, pp. 59–66, 1999.

[38] J. M. Pullen and E. White, "Dual-mode multicast: A new multicasting architecture for distributed interactive simulation," in *Proceedings of the 12th Distributed Interactive Simulation Workshop, Orlando, Florida*, March 1995.

[39] X. Shen, F. Bogsanyi, L. Ni, and N. D. Georganas, "A heterogeneous scalable architecture for collaborative haptics environments," in *2nd IEEE International Workshop on Haptic, Audio and Visual Environments and their Applications - HAVE*, September 2003.

[40] Sun Microsystems, Inc., "Overview of the java native interface." `http://java.sun.com/docs/books/tutorial/native1.1/concepts/index.html`, 2003.

[41] P. Karn and C. Partridge, "Round trip time estimation," in *ACM SIGCOMM*, August 1987.

[42] V. Jacobson, "Congestion avoidance and control," in *ACM SIGCOMM*, August 1988.

Appendix A

Organization of the Implementation

A.1 External Requirements

All the code of this program has been developed on Windows using Borland JBuilder, version 7. It requires the presence of the Java Development Kit (JDK). The source code has successfully been compiled with JDKv1.3.1 and JDKv1.4.2_01. It also requires Java3D. Version 1.2.1 has been used. Image capture in the tracheostomy simulation is done by writing JPEG pictures in the folder *C:\temp\toast*.

A.2 Folder Structure

The source code, java classes, required DLLs, documentation, and environment files have all been put in the same root folder, that is hereafter denoted by $rootfolder. This folder contains:

- Batch files that launch the different applications. Those batch files merely make sure that environment variables contains paths required by the java virtual machine. The variable *path* must contain "." (because some paths in the source code have been written relatively to $rootfolder), ".\bin" so that the JVM can find DLLs, and the path where the executable *java.exe* is located. *classpath* contains ".\classes" and the path to the jar files containing the Java3D classes;

- the playback architecture (folder systems);
- surgeryconfig.ini, which is a configuration file needed by the playback architecture;
- JBuilder files. It should be noted that JBuilder configuration is very unlikely to remain valid from one machine to another, and has therefore to be adapted (In JBuilder, select "Project" menu, then "Project properties" entry, and the "Paths" tab to do that).

A.2.1 Folders src and classes

The source folder contains the source code for the project. The packages toast, toast.transport, toast.util, toast.users, application.surgery, and cubetest are the core of the implementation. Three classes from the playback architecture had to be modified to fit the two applications. The source code of the modified classes is kept in src\systems. The source code for RTP/I, obtained from the RTP/I homepage, has been added to the project. After compilation, the corresponding class files are put in $rootfolder\classes.

The folder src\native contains the source code for DLLs that are intended to be used in the project. The only DLL that had to be generated is cubePhantomInterface. After DLLs are compiled using the source code in src\native, they have to be placed in folder $rootfolder\bin.

A.2.2 Folder content

This folder contains all resource files accessed by the project, and is also where generated files are put (except for images captured from the tracheostomy application).

content\config contains the network configurations. Those configurations are text files where 4 numbers separated by carriage returns are written. First number is the minimal loss rate of the network, second one is the maximal loss rate, third one is the delay and fourth one the jitter. Configuration files placed in any subfolder will not be accessed by TOAST.

content\logs contains the log files generated by TOAST. Each time TOAST is used, a new log file is generated in a folder named after the time in milliseconds at which the logs began. Relevant logs have been kept and manually sorted.

content\tracheostomy has the VRML descriptions of the operating room and patient, as well as JPEG pictures used for textures.

A.2.3 Other folders

$rootfolder\doc contains the documentation summarized in Appendix B, as well as the presentations and articles about this project. This thesis is available in doc as well. The API can be accessed via $rootfolder\doc\index.html .

$rootfolder\bin contains the specific DLLs required for the project. Those DLLs are needed to communicate with the GHOST API.

$rootfolder\bak is a backup of previous versions of the source code, generated by JBuilder.

$rootfolder\misc contains tools for generating a video from pictures, and the original source code for RTP/I.

Appendix B

Application Programming Interface Overview

The whole API is too voluminous to be printed here. The following table gives the complete list of all the classes of the implementation, with a short description. A checkmark indicates the classes implementing a main method. The name of interfaces is given in italics.

SharedObject		An active shared object in the 3D environment.
Toast		This class has to be extended by any application that wants to make use of this framework. It provides stream handling (automatically deciding which updates are key if the application does not do it) and loads the communication module chosen by the application.
UpdateMessage		The update messages exchanged between the users. The semantics of those updates are not provided. The number of degrees of freedom has arbitrarily been set to 6.

Table B.1: Package toast.

Bucket		**Deprecated** *Smoothed SCTP uses this bufferring mechanism and adds it to SCTP timely reliability.*
Comm		This class provides common methods for all the transport protocols and the interface of methods they have to implement.
LightTcp		The Light TCP transport protocol, which implements an updatable queue on the sender side.
Rtp_i		This class makes the link between TOAST and the RTP/I source code.
Sctp		Synchronous Collaboration Transport Protocol.
Smoothed_sctp		The smoothed SCTP transport protocol.
Udp		Represents the Uniform Datagram Protocol. Simple interface between TOAST and the network layer, that does not provide any functionality.

Table B.2: Package toast.transport.

TimerListener		Forces the classes using a timer to implement a method specifying the behavior when a timer times out.
Clock		**Deprecated**
ConverTools		Utility class that provides conversion from primitive types and IP addresses to byte arrays, and conversely.
logParser	✓	Small utility class written to analyze the logs generated by the box carrying and tracheostomy applications.
NetworkLayer		Simulation of the network layer impairments and interface between the transport layer and the network layer via the use of java DatagramSocket and java MulticastSocket.
NTPClient		NTP client.
SessionServer	✓	The session server for TOAST. Its role is to accept connections from every user of the session, and to send them the list of all the users already present.
StreamHandler		This class manages the interaction stream.
Timer		Simple Timer customized for use by the transport protocols.
WaitingList		Simple class that counts how much acknowledgment are still awaited.

Table B.3: Package toast.util.

ToolInterface		Interface implemented by all the classes that load and manage the tools.
COSMOSapplication		Serves as the general application that will make use of the COSMOS framework.
Gui		The graphical user interface of the tracheostomy application.
MuscleLayer		The Muscle Layer is a texture placed under the skin to provide a visual impression of what is under the skin.
Replay		Loads the update messages read in a log file and displays them according to their timestamp to provide a play back of a past simulation session.
Skin		This class creates a surgical area sensitive to the action of the tools.
SurgeryApp	✓	The main class of the Tracheostomy Application.
ToolManager		The tool manager loads and moves the surgical tools.
VideoCapture		This class reads a log file, recreates the tools movements and their effect on the skin. The movements are displayed in slow motion (to ensure that there is enough CPU) and frames are captured at a rate of 10 frames per second.
VrmlLoader		When the VRML loader of the playback architecture is called, the same methods are always called. This class has been written to simplify access to the loader itself.

Table B.4: Package application.surgery.

cubeCanvas		Extension of the Java3D canvas.
cubetest	✓	This class is the client side of the Box carrying application.
PhantomInterface		Interface with the PHANToM haptic device, devised for the box carrying application.
physicalCube	✓	Server side of the Box carrying application.

Table B.5: Package cubetest.