INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

Université d'Ottawa · University of Ottawa

Synchronous Collaboration in Virtual Environments: Architecture, Design, and Implementation

by

Shervin Shirmohammadi, M.A.Sc.

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

Doctorate of Philosophy
in
Electrical Engineering

Ottawa-Carleton Institute for Electrical and Computer Engineering

School of Information Technology and Engineering
University of Ottawa

Abstract

In terms of the distribution of objects' update information is today's Collaborative Virtual Environments (CVE), the focus in most existing architectures is on the movement of avatars and vehicles, not on collaboration. In this thesis, a novel architecture for the support of closely-coupled collaborative tasks in CVEs is presented. This architecture consists of an application-layer model that represents higher-level collaborative actions of users, and a communication architecture called the Synchronous Collaboration Transport Protocol (SCTP) which is a sender-initiated interaction-based transport protocol for CVEs. In addition to theoretical analysis and simulation, proof of concept is provided by implementation of the INVENTIST[1] prototype. The main goal of this research can be summarized as the design, justification, simulation, and implementation of an architecture that, in comparison to others, has higher efficiency for performing tightly-synchronous collaborative tasks in CVEs.

[1] INVENTIST: INception of Virtual ENvironments' TIghtly Synchronous Tasks.

Acknowledgements

I would like to thank my teacher and supervisor, Professor Nicolas Georganas, for his continuous support, motivation, and guidance during my close to five years of study and research at the Multimedia Communications Research Laboratory. Professor Georganas is considered among the best in the world in the field of Multimedia Communications, and I consider it an honor having had him as my supervisor. I will leave the MCRLab with great memories, in addition to my acquired knowledge and experience.

Special thanks go to my parents, to whom I dedicate this thesis, for their never-ending care and support, specially my mother who talked me into continuing my education to the highest level before getting into the work market.

I would also like to thank the following entities and acknowledge their financial support:

- the Natural Sciences and Engineering Research Council (NSERC) for their PGS-B scholarship (1998-2000);
- the Ontario Graduate Scholarship Program for their OGS scholarships (1995/96);
- the Telelearning Network of Centers of Excellence (TL-NCE) for providing the funds for various projects I was involved in during my stay at the MCRLab;
- the Canadian Advanced Technology Association (CATA) for their Telecommunications Software scholarship in 1998;
- IEEE Ottawa Section for their Annual Student Award in 1997;
- and the University of Ottawa Graduate Awards Office for their Excellence Scholarships (1996-2000).

Table of Contents

List of Abbreviations

CALVIN - Collaborative Architectural Layout via Immersive Navigation

CAVERN: CAVE Research Network

CORBA: Common Object Request Broker Architecture

COVEN: Collaborative Virtual Environments

CSCW: Computer-Supported Cooperative Work

CVE: Collaborative Virtual Environment

DIS: Distributed Interactive Simulation

DISCOVER: Distributed Information System for Collaborative Virtual Environments Research

DIVE: Distributed Interactive Virtual Environment

EAI: External Authoring Interface

HLA: High Level Architecture

ISD: Intra-Stream Delay

INVENTIST: Inception of Virtual Environments' Tightly Synchronized Tasks.

JETS: Java-Enabled Telecollaboration System

JSAI: Java Script Authoring Interface

LW: Living Worlds

MCRLab: Multimedia Communications Research Laboratory, University of Ottawa

MuTech: Multi-user Technology

NICE: Narrative Immersive Constructionist Collaborative Environments

OC: Open Community

OSS: Out-of-Synch State

OST: Out-of-Synch Time

RTI: Run Time Infrastructure

SCTP: Synchronous Collaboration Transport Protocol

UML: Unified Modeling Language

VE: Virtual Environment

VR: Virtual Reality

VRML: Virtual Reality Modeling Language

List of Figures

List of Tables

Chapter 1. Introduction

1.1 Collaborative Virtual Environments

The basic goal of virtual reality (VR) is to produce a simulated environment that gives to its user the impression that what he/she is experiencing in that environment is real. Imagine being able to view the latest Porsche model, see it from all possible angels, and have a test drive with your friend without even leaving your house. This is the idea behind virtual reality.

Many consider VR as the ultimate goal of communications: to overcome physical distance for all human perceptions [14]. Telephone overcame the distance barrier for audio, and Television did the same for video. Similarly, VR tries to eliminate this obstacle for all human senses including vision, hearing, touch, and even smell and taste. Virtual Reality may be considered to have been born in the mid-1960s, based on the work of Van Sutherland from the University of Utah. A paper, published in 1972 by D.L Vickers, one of Sutherland's colleagues, describes an interactive computer graphics system utilizing a head-mounted display and wand. The display, worn like a pair of eyeglasses, gives an illusion to the observer that he/she is surrounded by three-dimensional, computer-generated objects. The challenge of VR is to make those objects appear convincingly real in many aspects like appearance, behavior, and quality of

interaction between the objects and the user/environment.

Collaborative Virtual Environments (CVE), are shared virtual reality spaces where participants from different geographical locations can collaborate in performing tasks, as well as share data, as if really present at the same location. Audio conferencing capabilities further enhance the collaboration experience in such spaces and are in fact essential in creating a useful collaboration environment. There are numerous applications for CVEs in industry and society: Computer Supported Cooperative Work (CSCW) [19], collaborative design and engineering, collaborative scientific visualization, augmented reality (seeing real and virtual world at the same time) for sharing spaces, Teleimmersion [49], multi-user virtual workplace conferencing, arts and fashion design [48], virtual electronic laboratories, collaborative virtual tele-robotics, shared virtual environments for training, and games and entertainment.

There are many research topics regarding such CVEs. Traditionally, research in VR has focused on graphics: the development of natural interfaces for manipulating virtual objects and traversing virtual landscapes in a manner similar to the real world. Collaborative manipulation, on the other hand, requires the consideration of how participants should interact with objects and with one another in a collaborative manner. Some of the arising issues are: how participants should be represented in the collaborative environment, how to effectively transmit non-verbal cues that real-world collaborators use so effectively, how to best transmit video and audio both publicly and privately, and how to maintain a virtual environment even when all its participants have left [39]. On the technical side, the topics of interest are application sharing, avatars and participant representation, audio and video conferencing, session management, participant

awareness, event and data multicasting, conflict resolution, interactive and responsive objects, persistence, and many more.

1.2 Motivation

In terms of graphics and haptics, a tremendous amount of research has been performed over the past few years with some very good results. Using state-of-the-art 3D technology, it is now possible to represent and interact with sophisticated environments and objects such as workbenches [5] and cloth-like objects [25] in a convincing and natural manner. In addition, hardware technology required to run the sophisticated graphics of virtual environments has also advanced very rapidly and has become both more powerful and cheaper. The Sony PS-2 graphics chip, for example, is the latest in the series of this cheap and powerful hardware. Although currently used mostly in game consoles, chips and other technology such as the Sony PS-2 are believed by most experts to be the driving force behind bringing the concept of 3D environments out from labs and research facilities and into homes [3]. However, these same experts believe that there is still a long way to go in terms of IO hardware performance and ergonomics, interaction techniques, application software, and cost, before this move materializes.

One of the issues in collaboration in virtual environments is synchronous collaboration - the ability to allow multiple geographically distributed participants to perform closely-coupled tasks on shared objects. In addition to the graphics issue, synchronous collaboration requires also specific architecture and design at the application and the communication levels. Let us have a more detailed look at this issue:

The core technology behind any collaboration system is the exchange of **update messages**. An update message is generated as a result of a user's interaction with the shared environment. For example, when one user moves or rotates a shared object, an update message containing the new coordinates/orientation of that object is sent to all users in the system. The object representation in the other users' environments is then adjusted according to the update message received (Figure 1).

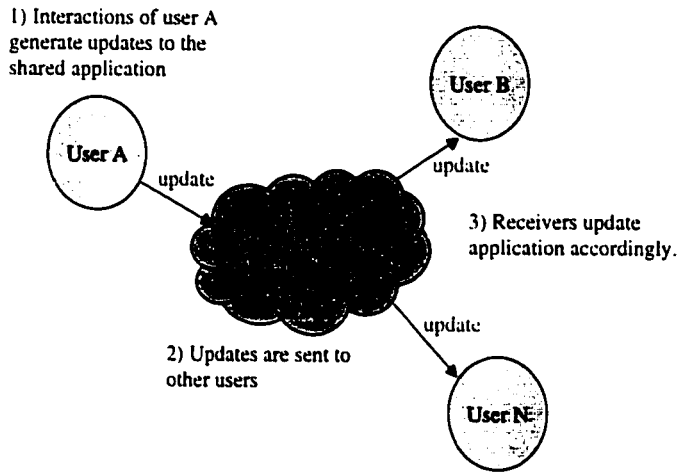


Figure 1. Exchange of update messages in collaboration systems.

This core technology is the 20% of the system that performs 80% of the work. The rest of the functionality provided by a collaboration system are for consistency and access control, time management, object registration, and other miscellaneous tasks.

How these update messages are exchanged depends on the implementation of the system. Many systems use a client-server architecture where a dedicated server is responsible for receiving and disseminating update messages among clients. Some systems use a fully distributed architecture where clients communicate directly with one another. Also, the transmission of the update message itself is implemented differently: some systems

construct network packets containing the update data and send the packet, while some others use mechanisms such as Remote Procedure Call (RPC) to invoke appropriate methods on other clients. Each of these architectures has its advantages and disadvantages, leading to different "qualities" of collaboration.

There are 2 categories of parameters that determine the quality of a collaborative session: application/graphics parameters, and communications parameters. At the graphics level, we have parameters such as realism of graphics, interaction quality, object behavior, and responsiveness of haptics such as VR goggles and gloves. The more responsive the equipment and the more realistic the graphics, the higher the quality of the environment. At the communications level, we have parameters such as end-to-end delay, jitter, and maximum number of supported simultaneous users.

All of these parameters are important. But two parameters are considered to be the most important parameters in terms of "collaboration": **delay**[1] and **jitter**. It is crucial in a collaboration space that an update message be received as quickly as possible, or the realism of collaboration will degrade to a degree that the session becomes "unnatural" and participants are unable to efficiently carry out their collaborative tasks. In fact, delay is the only parameter about which extensive performance studies have been carried out, in order to determine acceptability thresholds [22][26][30]. Jitter has recently been proven to be another very important parameter for collaboration, suggested to be even more important than delay [28]. Statistically speaking, jitter is the variance of delay. Practically, jitter causes irregularities in the chronological manner in which update

[1] In this document, delay is defined as the average amount of time that it takes for an update message generated by one user to reach another user in the system.

messages are received. However, unlike delay, no studies have been performed for jitter in order to determine a threshold as an acceptable level of jitter for collaboration. Jitter will be discussed further in chapter 4.

As we will see in the coming chapters, there are two major obstacles that render difficult the low-end-to-end-delay design of CVEs: reliability, and scalability (number of simultaneous users/objects supported by the system). We will also see performance evaluations of existing state-of-the-art shared VR systems. The results of these tests in general do not satisfy high-quality delay requirements for collaboration systems, mainly because all existing systems use available protocols such as TCP/UDP and multicast which are generic protocols and have not been designed to meet the specific needs of CVEs. Based on the inadequacy of the available systems, and further analysis of collaborative update messages in CVEs, an architecture is proposed which is specifically designed for synchronous collaboration in CVEs, and takes into account the characteristics of update data transmitted in a collaborative space.

1.3 Objectives

The Multimedia Communications Research Laboratory (MCRLab) and the Sensing and Modeling Research Laboratory (SMRLab) at the University of Ottawa have recently launched a major project in Collaborative Virtual Reality, called DISCOVER (Distributed Information System for Collaborative Virtual Environments Research). In partnership with the industry (Newbridge, NuVision), government (CRC) and Centers of Excellence (CITO, CITR, TL-NCE), the project will address CVE applications for

training and telecollaboration.

In the context of the DISCOVER project, some CVE prototypes have been developed at the MCRLab. These prototypes are based on widely available standards such as VRML, Java3D, RTI [61], IPv6, and SPLINE [42].

My goal is to come up with a CVE framework that is more efficient than what the existing collaborative systems use in terms of synchronous collaboration, and utilize the framework in the prototypes developed at the MCRLab. This architecture must meet the collaboration-specific requirement of tightly-coupled collaborative actions, which we shall see in the coming chapters.

1.4 Contributions

This work introduces novel ideas and concepts not considered by existing work, including:

- Study and consideration of *collaborative* update messages, as opposed to regular update messages studied by other work. In this work, new and previously-unperformed research is being done specifically regarding synchronous collaborative updates in CVEs. This leads to an Architecture proposed for synchronous collaboration tasks in CVEs, based on the action of synchronous collaboration in the real world.

- A novel perspective that considers collaboration data as *Interaction Streams*, each consisting of a burst of update messages with a final and critical update message. This

perspective will enable performing closely coupled collaborative tasks in Virtual Environments.

- A representative-based sender-initiated timely-reliable communication architecture for key update messages, specific for CVEs.

- A transport protocol which is more efficient than the existing ones in terms of synchronous collaboration requirements.

One customary measure of contribution is publication, which indicates recognition of one's work by the scientific community. The following is a list of the papers that I have published during my Ph.D. program. All papers are related to Synchronous Collaboration technology, either in the area of traditional CSCW or the more recent field of CVE. Publications preceded by a diamond (♦) are directly related to this Ph.D. thesis.

JOURNALS:

1- S. Shirmohammadi, A. El Saddik, N.D. Georganas, and R. Steinmetz, "JASMINE: A Java Tool for Multimedia Collaboration on the Internet", Journal of Multimedia Tools and Applications (MMTA), 2000 (accepted, to appear).

♦ 2- S. Shirmohammadi and N.D. Georganas, "An End-to-End Communication Architecture for Collaborative Virtual Environments", Journal of Computer Networks (accepted, to appear).

3- S. Shirmohammadi, Li Ding, and N.D. Georganas, "An Approach for Recording Multimedia Collaborative Sessions: Design and Implementation", MMTA Journal (submitted).

4- S. Shirmohammadi, A. El Saddik, N.D. Georganas, and R. Steinmetz, "Web-Based Multimedia Tools for Sharing Educational Resources", ACM Journal of Educational Resource in Computing (submitted).

MAGAZINES:

5- S. Shirmohammadi, J.C. Oliveira, and N.D. Georganas, "Applet-Based Multimedia Telecollaboration: A Network-Centric Approach", IEEE Multimedia, Volume 5, Number

2, April-June 1998, pp. 64-73.

CONFERENCES:

6- S. Shirmohammadi, J.C. Oliveira, and N.D. Georganas, "Java-Based Multimedia Collaboration: Approaches and Issues", Proc. IEEE/IEE International Conference On Telecommunications (ICT '98), June 1998, Vol. I, pp.127-131. (invited)

♦ 7- J.C. Oliveira, S. Shirmohammadi, and N.D. Georganas, "Collaborative Virtual Environment Standards: A Performance Evaluation", Proc. IEEE Workshop on Distributed Interactive Simulations and Real-Time Applications (DIS-RT '99), pp. 14-21.

8- J.C. Oliveira, S. Shirmohammadi, and N.D. Georganas, "A Collaborative Virtual Environment for Industrial Training", Proc. IEEE Virtual Reality (VR 2000), p.288.

♦ 9- S. Shirmohammadi and N.D. Georganas, "An Architecture for Collaboration in Virtual Environments", Proc. IEEE Virtual Reality (VR 2000), p.283.

10- S. Shirmohammadi, A. El Saddik, N.D. Georganas, and R. Steinmetz, "JASMINE: Java Application Sharing in Multiuser Interactive Environments", Proc. 7th International Workshop on Interactive Distributed Multimedia Systems and Telecommunication Services (IDMS 2000), October 2000.

11- J.C. Oliveira, S. Shirmohammadi, et al, "Virtual Theater for Industrial Training: A Collaborative Virtual Environment", Proc. 4th World Multiconference on Circuits, Systems, Communications & Computers 2000, pp. 294-299.

♦ 12- S. Shirmohammadi and N.D. Georganas, "Collaborating in 3D Virtual Environments: A Synchronous Architecture", Proc. IEEE International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE 2000), pp. 35-42.

It is worthwhile to mention that publication number 12 received the **Best Paper Award** at the WETICE 2000 conference in the Knowledge Media Workshop.

1.5 Outline

The rest of this thesis is organized as follows: chapter 2 familiarizes the readers with some background about CVEs and the data exchanged in such environments. Chapter 3 presents some state-of-the-art and also some classical VE systems that are also related to

CVEs, as well as a critical analysis of these systems. Next, chapter 4 introduces the theory of the proposed architecture and design both at the application and communication level. Then, chapters 5 presents the simulation and test parameters chosen, and analyzes the results, while chapter 6 presents the INVENTIST framework and analyzes the subjective test results. Finally Chapter 7 concludes this thesis.

Chapter 2. Background

There are different types of data used in a CVE. This is the result of the vast user capabilities in a virtual environment: users can walk around, interact with objects, interact with one another, chat through text or audio, and even video-conference. Each of these types of data has its own characteristics in terms of requirements such as reliability, delay, and bandwidth. This is discussed next.

2.1 CVE Data Types

Data exchanged in a CVE can roughly be categorized as follows:

- **real-time audio and video data**: data carrying the video or audio streams in the environment.

- **object/scene description data**: data describing an object and its attributes; e.g., description of an avatar.

- **CSCW data**: this represents traditional 2D collaborative data such as whiteboards.

- **control data**: used by the system to perform tasks such as consistency control, management, and so on.

- **update messages**: such as avatar/vehicle motions.

- **collaborative update messages**: update messages representing a collaborative task. This type of data is the focus of this work and will be explained in great detail shortly.

Audio and video data have well-known characteristics. Extensive studies have been done regarding optimum ways to transmit such data in real time. As a result, standards and protocols such as the Real Time Transfer Protocol (RTP) have been developed and successfully used to handle the requirements of real-time audio/video streams. Thorough discussions about this type of data is beyond the context of this thesis.

Object/scene data are required by a CVE when a new type of object is being introduced into the environment, or when a user "jumps" from one environment into another. For example, a user might wish to use a custom avatar as opposed to an avatar predefined by the CVE system. In this case the system must download the description of the custom avatar. Another example is when a user brings or inserts a new object into the environment; the description of that object must again be downloaded. This type of data is not very frequent and usually does not have any strict delay requirements. It does however require reliability and cannot afford to lose any portions of the data.

CSCW data usually require both low delay and high reliability. Similar to live conferencing, CSCW has been subject of extensive studies [19][52]. In general, CSCW update packets cannot afford to be lost for obvious reasons, and they also have strict latency requirements ranging from 200 msec for high quality sessions to 1 second for minimum acceptable sessions [38]. As a result, CSCW data are usually transmitted by Reliable Multicast (RM) protocols [11][46], or client-server based architectures.

Control data are similar to CSCW data in that they need both low delay and high reliability. The difference between the two types is that control data is sent less frequently than CSCW data.

Update messages have both strict delay and reliability requirements. Extensive studies have been performed regarding update messages in virtual environments and different protocols and techniques have emerged as the most widely utilized standards, such as Distributed Interactive Simulation (DIS) [16].

Collaborative update messages, the main subject of this thesis, also have strong delay requirements, and are also severely affected by jitter. Note that there is a distinction between collaborative update messages and regular update messages. A collaborative update message is one that signifies a tightly coupled action, such as update messages corresponding to the position of a chair which is being carried by two remote users simultaneously, each holding one end of the chair. A regular update message is, for example, the motion of a by-passer's avatar which, although important, is not critical for collaborative purposes. One contribution of this work is the demonstration of the fact that collaborative update messages are somewhat different from regular update messages and require a class of their own. Also notice that there are two modes of collaboration: synchronous mode and asynchronous mode. Unlike asynchronous collaboration which doesn't require very tight thresholds in terms of networking issues, latency and reliability heavily affect synchronous collaboration. Each of these modes of collaboration has its own set of applications and they are not competing technologies in most cases, rather complementary. This work focuses on synchronous collaboration and not the asynchronous type. In this document, the term "update messages" refers to the

synchronous collaborative update messages, unless otherwise indicated.

2.2 Requirements for Collaboration

It has been suggested that good collaboration environments must have an end-to-end delay of no more than 100 msec for update messages [30]. Other studies have loosened this requirement to 200 msec as "acceptable" delay [26]. The reason for such strict requirement is the way collaboration works in the real world. Collaboration is a real-time reactive multi-user process: many users work on something together, making real-time decisions based on the actions of each other towards achieving a common goal. For example, assume two people are to carry a heavy box. In the real world they must synchronize their actions such that they lift the box more or less at the same time. If one person lifts one end of the box and another person doesn't react quickly enough to lift the other end, the first person might have to drop the box and break it, or get injured trying to hold it. The above mentioned delay constraints achieve this objective and make collaboration possible in VR.

Jitter is another important parameter for collaboration. It has been shown that a CVE session with a low 10 msec delay with jitter results in a collaboration environment which is almost as bad as one with 200 msec delay but no jitter [28], for collaborative purposes. Nevertheless, no clear figures have yet been found to indicate what is acceptable jitter and what is not. At this point, the goal is to minimize jitter as much as possible - a task which is quite hard to achieve from a transport layer perspective. Let us see why.

Jitter usually occurs because at different times, the network has different states. Routers

have a certain buffer capacity and processing delay, leading to variations in how long a packet spends time in a queue. Sometimes routers drop a packet, which later has to be retransmitted, causing additional delay. Also, links have a maximum bandwidth and propagation delay. In addition, a given packet might take different routes to reach the same destination as some other packets; this is specially frequent in the case of the Internet. As a result, a packet sent at time t_1 might spend more time or less time in the network compared to a packet sent at time t_0 or t_2. It is obvious that it is very hard to address the jitter problems at the transport layer since it is mostly a physical layer issue.

In addition to delay and jitter requirements, update messages have also a strict reliability requirement for the same reasons explained above. It is obviously important that all users receive update messages. If one person doesn't receive an update message, he or she will be out of synch with others and cannot collaborate efficiently. The question is: do we need to achieve reliability for all update messages or only for some of them?

This question can be answered by realizing the fact that the last state of a shared object is the most crucial data in a CVE. For example, a user might move an object on a table by 40 centimeters. Assume that this interaction creates 20 update messages each 20 msec apart and each representing the position of the object at a different time. The duration of the whole interaction is less than half a second (400 msec). Further assume that update messages 1, 12 and 17 are lost. In this scenario, it is clear that we do not need to retransmit the lost messages since the last message which indicates the final position of the object was received. It would be very inefficient if a receiver which detects a lost message asks for retransmission, because old messages are obsolete and the latest state is what hosts are interested in. In addition, in a Wide-Area Network (WAN) environment,

there's a good chance that the delay requirement is violated by the time the lost message is recovered. Therefore, a receiver should not ask for lost messages as long as it has the latest state of a shared object. This property of update messages in CVEs has been discussed in other works as well [12] and it is the main reason why reliable multicast protocols, which recover lost messages at the expense of higher latency and network bandwidth, should not be used in CVEs. A more detailed discussion about reliability will be presented in chapter 4.

2.3 Protocols for CVEs

A superficial analysis of the above data types confirms that no single one protocol can be used to accommodate the requirements of all data types. This leads to the conclusion that a CVE communications stack must be comprised of a **hybrid protocol** that is composed of different protocols for different data types.

As mentioned earlier, a protocol such as RTP is suitable for real-time audio and video transmissions. For object/scene description data, which is not transmitted frequently, does not have strict latency requirements, and requires reliability, a protocol such as TCP or HTTP can be used. Using any of these protocols, a VR system can download object or scene description when necessary.

There are many architectures used to transmit CSCW data. Some systems use a centralized server, while others are fully distributed. In the server-based system, one-to-one TCP connections are used where each message is reliably transmitted to all users one by one by the server. In distributed architectures, the protocol of choice is usually some

type of reliable multicast protocol. Other combinations of architectures and protocols also exist. Each of these architectures has advantages and disadvantages when compared to others. However, detailed analysis and discussion about each of these architectures is beyond the scope of this document.

Control messages can also benefit from the same architecture used by CSCW data, as they both have very similar requirements.

For regular update messages, the most commonly transmitted type of data in CVEs, standards such as DIS have been used for many years with some degrees of success. Generally speaking, this data type has a strict latency requirement, but is flexible against network loss due to the usage of techniques such as dead-reckoning and stay-alive transmissions. These techniques are briefly explained in chapter 3.

This brings us to collaborative update messages. They have strict latency requirements, suffer from jitter, and require special reliability. Update messages are the focus of this thesis and are analyzed in chapter 3 where it will be shown that this type of data needs its own transmission protocol.

Hence, it becomes apparent that a Collaborative Virtual Environment must use a hybrid protocol to support the variety of data types used.

2.4 Summary

Table one on the next page summarizes the discussions in this chapter. Notice that the reliability requirement for collaborative update messages has been marked as "special".

We will see more about this in chapter 4.

Table 1. Summary of data types and their requirements in CVEs.

Data Type	Latency Requirement	Reliability Requirement	Suitable Protocol
real-time audio/video	strict	flexible	RTP
object/scene description	flexible	strict	TCP, HTTP
CSCW data	strict	strict	client-server/ reliable multicast
control data	strict	strict	client-server/ reliable multicast
update messages	strict	flexible	DIS
collaborative update messages	strict	special	-

Chapter 3. Current State of the Art

The specific research presented in this thesis is the design, simulation, and implementation of an architecture for the support of tightly-coupled collaborative update messages in CVEs. Before describing this proposed architecture in any detail, a brief overview of related work is necessary.

3.1 Popular VEs

Let us take a look at some of the existing VE systems of today.

3.1.1 The Distributed Interactive Simulation (DIS)

This is perhaps the most famous standard for distributed simulations. The Distributed Interactive Simulation (DIS) came into existence as a U.S. government initiative to define an infrastructure for linking entities of various types at multiple locations to create realistic, complex, virtual worlds for the simulation of military battles. DIS exercises are intended to support a mixture of virtual entities with computer controlled behavior, virtual entities with live operators, live entities, and constructive entities. DIS draws heavily on experience derived from the Simulator Networking (SIMNET) program

developed by the Advanced Research Projects Agency (ARPA), adopting many of SIMNET's basic concepts and heeding lessons learned. In short, DIS provides for the capability to set up battlefield scenarios, with both friendly and opposition forces, on a specified terrain.

Of interest to this research is the mechanism of propagating the update messages or Protocol Data Units (PDU) as they are called in DIS terminology. DIS uses multicast UDP packets to propagate state information about an entity among hosts. Obviously the network traffic is greatly reduced by using multicast because the sender transmits only one packet to the whole multicast group. Yet studies have shown that network traffic is still quite high with many simultaneous users in one simulation.

To overcome this deficiency, DIS uses a technique called *Dead Reckoning*. The idea behind dead reckoning is that instead of just sending an entity's location, a host sends a message that contains the entity's location, a time stamp, and a velocity vector. Using that information, each host in the network can then predict the entity's future locations based on the velocity without additional updates. Of course once the velocity vector changes, the prediction becomes erroneous. DIS addresses this issue by having each entity run the dead reckoning algorithm for its own object. As soon as the difference between an actual location and a predicted location exceeds a certain threshold, the entity re-transmits an update message with current position/velocity. This technique reduces the network traffic significantly when objects are short-term predictable.

In addition, DIS entities send a periodic update message called *keep-alive*. If this message is not received from an entity within a given duration, it is assumed the entity's existence

in the simulation has been terminated. Moreover, if someone enters the simulation after it has started, they will receive every entity's state within a few seconds through the keep-alive messages.

Despite all these techniques, experience has shown that large simulations still create a bandwidth problem. In addition, the processing of update messages for a large number of entities overwhelms hosts. This is due to the fact that all entities in the simulation transmit update messages to all hosts, even if some hosts are not interested in some entities (because they are too far apart, for instance). So DIS introduces yet another technique: *filtering*. Filtering causes unwanted update messages to be dropped at the host without processing. But this means that the update messages are still transmitted throughout the network.

3.1.2 High Level Architecture (HLA)

In accordance with the U.S. Department of Defense (DoD) Modeling and Simulation Master Plan, the Defense Modeling and Simulation Office (DMSO) is leading a DoD-wide effort to establish a common technical framework to facilitate the interoperability of all types of models and simulations among themselves. This Common Technical Framework includes the High Level Architecture (HLA) [61]. In December 1997, HLA was accepted as a draft IEEE standard to be supported by the Simulation Interoperability Standards Organization (SISO). The High Level Architecture is composed of three parts: the HLA Rules, the HLA Interface Specification, and the Object Model Template.

The HLA Rules document describes the general principles defining the HLA, and

delineates ten basic rules which apply to HLA federations and federates. The HLA Interface Specification defines the functional interface between federates and the Runtime Infrastructure (RTI). The Object Model Template Specification provides a specification for documenting key information about simulations and federations. In short, HLA attempts to provide a very generic environment that any virtual object can attach to in order to participate in a simulation.

RTI is the part relevant to this research. The Runtime Infrastructure contains the communications entities that propagate update messages among entities in the virtual environment. HLA does not specify how exactly the underlying communications module of RTI must be implemented, but it does specify certain requirements that it must meet. HLA specifies that RTI must support two low level services: "best effort" and "reliable". Existing implementations of RTI use UDP multicast to implement the best effort service, and TCP channels for the reliable service.

An efficient technique used in HLA is the concept of federations. Entities (called federates) can join federations of their choice and therefore only receive update messages from those federations. This greatly reduces the number of update messages exchanged among entities since a federate does not receive update messages that it is not interested in.

3.1.3 Open Community and ISTP

Open Community (OC) is a standard for multiuser enabling technologies from Mitsubishi Electric Research Laboratories, based on its previous work on Scalable Platform for

Large Interactive Networked Environments (SPLINE) [42]. Open Community is a software library, callable from ANSI C or Java, designed to provide many of the essential services necessary to make real-time multi-user cooperative environments possible. Open Community takes care of issues like network communications, real-time audio transport, application-neutral transport of large or complicated objects such as VRML models, and region-of-interest filtering.

A big advantage of OC over DIS is the idea of partitioning the world into *locales* [17]; each application's copy of the world model contains only those locales that are relevant to the application's current state. There is no need for any application to have an all-locales copy of the world model or to receive update messages from entities outside of its locales of interest. For example, a person playing cards in his room doesn't need to know the location of a fish swimming in the sea. This is the primary way that OC achieves scalability.

For its communications, SPLINE uses the Interactive Sharing Transfer Protocol (ISTP) [43]. ISTP is a hybrid protocol that supports the sharing of information about a virtual world among a group of user processes. It uses multiple protocols in order to support different data types in CVEs:

- TCP is used for the reliable communication of control information.

- UDP and RTP are used for the communication of time critical information such as update messages. (UDP and RTP messages are both sent via multicast whenever possible.)

- HTTP is used for the distribution of large pieces of data, such as object/scene description data.

ISTP uses the following architecture to achieve reliability: each locale has a server and each process in a locale has a one to one TCP connection with that server. Update messages are sent by UDP multicast, but transmission of acknowledgments as well as lost update messages is done on a one-to-one basis between each process and the server [42].

3.1.4 DIVE

The Distributed Interactive Virtual Environment (DIVE) is an internet-based multi-user VR system developed at the Swedish Institute of Computer Science [41]. Participants of a DIVE session navigate in 3D space and see, meet and interact with other users and applications. DIVE is an experimental platform for the development of virtual environments, user interfaces and applications based on shared 3D synthetic environments. It is especially tuned to multi-user applications, where several networked participants interact over a network.

DIVE is based on a peer-to-peer approach with no centralized server, where peers communicate by reliable and non-reliable multicast. By using different multicast addresses, it is possible to form groups and exchange data among entities in a group.

3.1.5 CAVERN

CAVERN, the CAVE Research Network, is an alliance of industrial and research institutions equipped with CAVE, ImmersaDesks, and high-performance computing

resources all interconnected by high-speed networks to support collaboration in design, training, scientific visualization, and computational steering, in virtual reality. CAVERN is a continuation and generalization of the work done at the Electronic Visualization Laboratory on several collaborative virtual environments in recent years. Two of these environments are CALVIN [21] and NICE, which provided testbeds to prototype several of the ideas that would eventually form parts of CAVERNsoft.

CAVERN has allocated a tremendous amount of research and development in user experience and graphics. It uses highly expensive specialized rooms (known as caves), dedicated multiprocessor computers for rendering, VR equipment, and high speed optical networks to create immersive virtual environments. For its communications module, CAVERN uses an Information Request Broker (IRB). In concept, IRB is very similar to the Common Object Request Broker Architecture (CORBA) in that it acts as a common bus between entities in the environment. Using the IRB, a client can arbitrarily form a channel, after having acquired the proper permissions, with any other client or server to access its resources. Clients may request 3 types of channels: reliable TCP, unreliable UDP, and multicast.

3.1.6 NPSNET

A famous paper by Macedonia et al. introduced the idea of partitioning a virtual environment into different *areas of interest* and using a separate multicast address for each area [34]. An entity would then only register in a few areas and consequently receive update messages from objects that it is interested in. Used in the NPSNET prototype, this approach proved to significantly reduce the host processing and network

bandwidth requirements compared to DIS.

This concept is very similar to locales used in Open Community. It seems that they were developed independently at approximately the same time.

3.1.7 MR Toolkit

The MR Toolkit is one of the older systems supporting virtual environments [55]. It is mentioned here only for historical reasons, as it used to be popular in 1993. This system uses a peer-to-peer reliable messaging system. An entity sends its update message to each and every other host in the system, one by one. This approach yields $O(N^2)$ update messages and does not scale well at all.

3.1.8 BrickNet

BrickNet is a client-server toolkit that allows sharing of object geometries as well as behavior [13]. Each client is connected to a server which handles many clients. Servers are able to communicate with each other to satisfy client requests.

3.1.9 Selectively Reliable Transmission Protocol (SRTP)

Though not a framework collaborative virtual environment, SRTP is a communications protocol that can be used to disseminate update information in such environments. Suggested as an Internet Draft in 1997, this hybrid protocol has 3 modes of operation [33]:

- **Mode 0**: for data that does not require reliability. For this mode, SRTP uses a multicast architecture.

- **Mode 1**: for data that must be received reliably by all members. For this mode, SRTP uses a NACK-based reliable multicast protocol, with a "NACK suppression" mechanism as an effort to avoid the *NACK implosion* problem [56].

- **Mode 2**: for data that must be received reliably by a single known member in the group. The assumption is that in this mode, an entity requires reliable communication between itself and only one other entity. For this mode, SRTP uses either TCP or ACK-based multicast, with non-participating entities ignoring the ACK message.

3.2 Communications Aspect: Critical Analysis

In terms of communications, we can categorize most of the above systems into 5 groups based on their topology and architecture; those that use hybrid protocols might fall into more than one category:

1. **one-to-one communication**: In this architecture, an entity sends update messages to each of the other hosts in a peer-to-peer fashion. It is obvious that this approach becomes inefficient with increasing number of entities and hosts and therefore does not scale. Examples: MR Toolkit, VEOS [57].

2. **client-server**: A client-server architecture transfers the task of sending update messages from a client to a server. A client sends its update message to a designated server which then forwards it to other clients. The advantage of this design is that a

server can decide which clients are interested in a given update message and send the update message to those interested only, reducing the processing time and bandwidth compared to the peer-to-peer approach. However, the server becomes a bottleneck in this architecture because it has to process messages coming in from many clients. Scalability is therefore achieved at the expense of more computing and network resources (more servers). Furthermore, there is an additional delay created since each update message must go to a server first, be processed, and then sent to its destination(s). Therefore, this approach also has a latency problem. Examples: BrickNet, RING [54].

3. **broadcast**: In this architecture, an entity sends its update message as a single broadcast packet which is received by everyone on the subnetwork. At first glance, broadcasting seems to solve many of the above problems since it is fast and only one messages is sent per update message. However in practice broadcast turned out to be a nightmare because every host on the subnetwork, even those not participating in the simulation, receive and process the broadcast message. As a result, these types of systems have been only used with powerful computing stations and very high bandwidth optical networks. Examples: SIMNET, VERN [4].

4. **multicast**: Multicast improves upon the lessons learned in broadcast. In this architecture, only hosts that are registered with the multicast address receive update messages. The most successful implementation of this approach was the DIS standard which was discussed in detail in 3.1.1. However, there are many significant problems with DIS when it comes to collaborative update messages. First, there is no guaranty that the last state of a shared object is received by all participants. This is a direct

result of IP multicasting - there is no reliability. Keep-alive messages, which are transmitted every 5 to 10 seconds, recover the last state of an object; however, this falls far short of the 100 to 200 msec latency requirement of collaborative environments, not to mention how much bandwidth these keep-alive messages waste since an object must send keep-alive messages even when it is not changing its state.

5. **reliable multicast**: The advantage of RM protocols seems to be twofold: they use multicast while at the same time they assure reliability for all messages. But this very fact makes them inefficient for CVEs: RM achieves reliability for all transmitted packets, something which is not required for all update messages in a CVE and creates undesired extra bandwidth and delay.

From the above architectures, only partitioned multicast systems in their reliable mode seem to be appropriate for collaborative update messages CVEs. Even though we don't need reliability for all update messages, the fact that we partition the environment into more manageable groups and have fewer people in each group, might justify fully reliable architectures. But reliable service always comes at the price of higher latency which might not satisfy the delay requirements. The question is how much is the latency associated with these systems? At the MCRLab, we designed and conducted the following experiments to evaluate the latency characteristics of the SPLINE prototype (Open Community) and the RTI system[1].

[1] Credit of performing these experiments goes to myself, Shen Xiaojun, and Jauvane Oliveira.

3.2.1 Performance Evaluation

At the MCRLab, we have developed several training and other CVE applications. One of them is a training application where an instructor teaches one or more trainees how to configure a specific ATM switch. This application requires coordination of interactions with different components of the ATM switch. In this example the shared objects are the ATM cards that must be inserted or removed for configuration. A screen shot of a sample session is shown in figure 2.

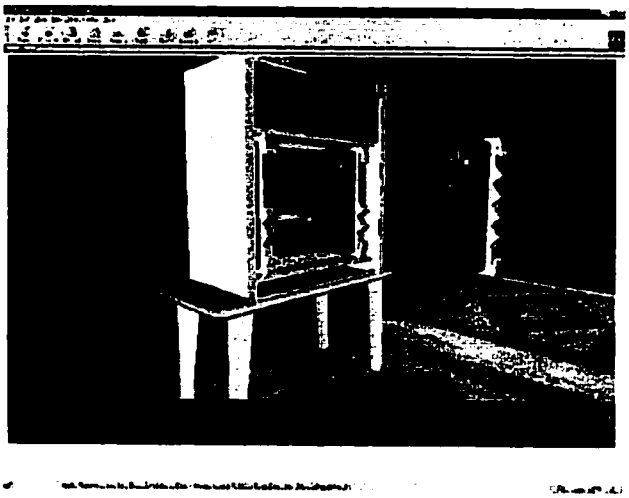


Figure 2. A Virtual ATM switch with one of its cards pulled out, running in Netscape Navigator.

The figure shows the virtual training environment running in Netscape Navigator equipped with a VRML 2.0 plug-in. To join the session, participants navigate to a pre-defined URL address. The browser then downloads the necessary HTML files, containing the Java applets and the VRML world. The VRML world is controlled by a Java applet trough VRML's External Authoring Interface (EAI). Since both SPLINE and RTI 1.0 release 3 provide a C API at this time, the Java Native Interface (JNI) was used to

connect the controlling Java applet to the communications middleware. This is further illustrated in Figure 3.

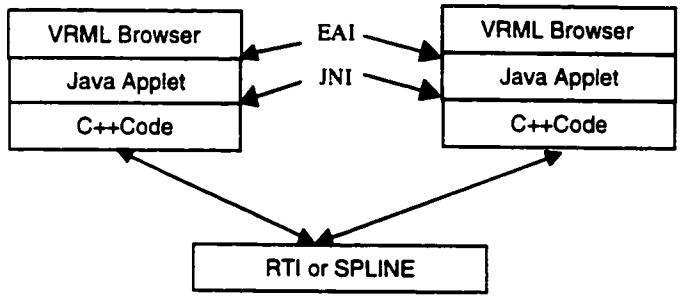


Figure 3. The Spline-based hybrid system model.

In our evaluations, we combined simulation and network delays as the *client-to-client delay* (CCD). This is the average time it takes for a given update message at one client to be assembled by the simulation, sent over the network, received by another client, and disassembled to be processed by the graphics/display components of the system - it doesn't include the rendering and graphics delay. In the case of the RTI prototype, for example, this delay includes all layers of Java, JNI, C++, RTI, and physical network delays. We also measured the middleware delay itself.

For the CCD test, we had an "initiator" client create an ATM card, and perform a translation on it (not at the graphics level). A second client then receives this update and extracts its information. Then, the second client performs a similar translation on the card which is "seen" by the initiating client. The initiating client again performs an update on its object , and so on. This procedure is repeated for a given number of times or duration, which was about 10 minutes in our tests. RTI sends out update messages automatically, but SPLINE allows us to specify how often update messages are sent. For the tests, we

tried 50 msec intervals (20 updates/second) and 1 msec intervals (1000 updates/sec).

All tests were performed on Pentium II 400 MHz PCs running Windows NT 4.0 and connected to a 100 Mbps Ethernet with 7 workstations. The graphics was set at 1280X1024 24bit color screen resolution with 3D accelerator cards. All machines were running typical operating system and networking background processes. The results are shown in the table below:

Table 2. Performance results for various middleware(All delays are in msec)

	RTI with VRML	RTI with Java3D	Spline with VRML 50ms update	Spline with VRML 1ms update
CCD	184	152	194	98
Middleware delay*	103	103	149	44

* This is CCD minus the delays between the middleware and the rendering part (VRML/Java3D).

Notice that these tests were performed between two users only, and on a 100 Mbps Ethernet with a *ping* time of less than 1 msec among all stations[1].

Considering the accuracy of the test, the first observation we can make is that the 100 msec requirement is barely met by RTI. However, taking into account how little traffic the 100 Mbps subnetwork was carrying, and that only two users were involved in the simulation, the numbers are unsatisfactory. As for SPLINE, we can see that the latency requirements are violated even at 20 updates per second which is a high rate of update. Since in this "ideal" set-up RTI just meets the latency requirements and SPLINE fails it, it can be concluded that if we were to support a large number of participants over a wide

[1] The ping test returned an average of zero msec, which means that the delay is smaller than 1 msec.

area network, these systems would not meet the required CVE delay thresholds and therefore fail. As argued earlier, this high latency was expected as a byproduct of supporting full reliability.

From the above results, we can conclude that these standards and systems leave a lot to be desired when it comes to collaboration [23]. In general, the available systems today focus on general distributed simulation scenarios and are too generic for collaboration specific environments. The general assumption in distributed simulations is that objects transmit update messages often, and that the latest state of things can be determined by techniques such as dead-reckoning algorithms because they are somewhat predictable. Experience has shown that these assumptions work very well for scenarios such as simulation of battlefields or multi-user avatar-based games. In fact most of these systems have been specifically designed for either military purposes or games and they do a good job at that. In shoot-em-up games or battlefield scenarios, people, tanks, planes, and other war machines are almost constantly moving in a short-term predictable manner. A plane's course of flight can be extrapolated from its position and velocity vector. Also, a lost update message is usually followed by many other update messages, or keep-alive messages.

However, these assumptions fail for CVEs. In fact in CVEs the conditions can be quite the opposite: shared objects often do not send continuous update messages, and when they do it is not necessarily in a predictable manner. When coordinating closely-coupled collaborative tasks in CVEs, there is no room for "guessing" the state of a shared object. All participants must reliably receive the most current state, or collaboration might fail. It is a well-known fact that dead-reckoning and similar algorithms don't guarantee that all

hosts share identical state about a shared entity and require hosts to tolerate discrepancies; and that they lose efficiency in situations when precision and consistency are vital [53], such as collaborative situations.

One of the reasons existing systems fail to fulfill collaboration needs is that they do not consider the properties of collaboration data itself. This is evident from the number of literatures which spell out the requirements for shared virtual environments, but don't look at collaboration data specifically[32][46][51][55]. In the next chapter, we will see that CVE collaboration has its own data characteristics which sets it apart from other types of data and requires special handling.

Chapter 4. Theory: The Proposed Architecture and Design

This chapter is an attempt to explain in detail the theoretical aspect of my proposed architecture, and why it is more efficient for synchronous collaboration in CVEs than existing architectures. The discussion starts with some observations of synchronous collaboration scenarios and how they are actually carried out by people.

4.1 A Closer Look at Collaboration Data

To have a better understanding of the characteristics of collaboration data exchanged among entities, let us have a look at some typical collaboration scenarios.

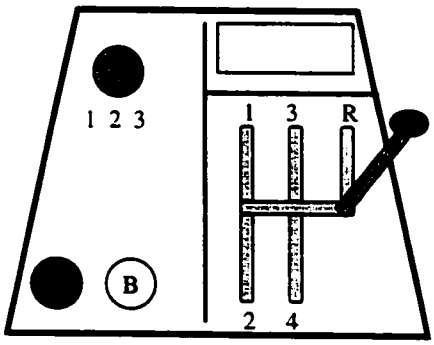


Figure 4. A crane console with buttons, knobs, and handles, in a training application.

Figure 4 illustrates a console which in this example is a simulation of a console of a construction crane. Assume a trainer is teaching many participants how to operate the crane. The types of operations performed in this scenario are very short motions of many pieces of gadgets. For example, the instructor says: "To lift up a load, press button A, put the knob in position 3, and move the handle to position 2 while holding down button B. Mr. Smith, please show me how you would do that." Now, Mr. Smith is going to perform all of the above tasks on his virtual training system and the trainer will observe his actions. The following typical update messages are generated:

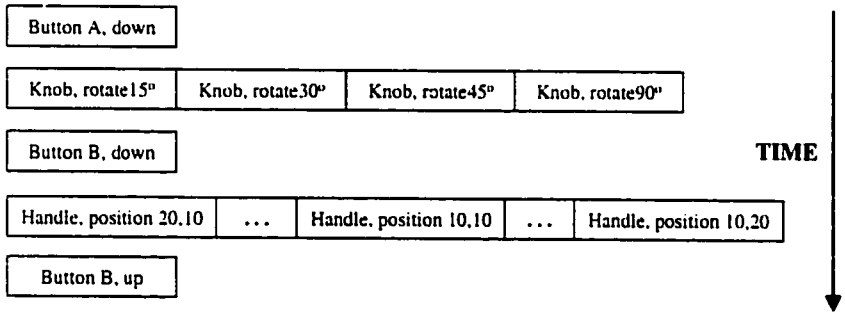


Figure 5. Typical update messages generated.

The exact number of messages depends on the application itself and the graphics system, but we can make the following general observations:

1) Each interaction creates a rather short *stream* of update messages, sometimes only one update message.

2) These streams occur in very short "physical" times, often less than a few seconds, and the coordination/orientation of the shared objects are not predictable since one might choose many of the possible different ways to move the handle, for instance. In addition, the duration of each action is so short that using a

prediction algorithm is simply not justifiable.

3) The most important message, especially in short streams, is the last message. For example, if the first and third message of the knob stream are lost, the result will be a less smooth animation; but if the last message is lost, the result will be a collaboration failure since the last position of the knob will be different on the screen of different participants, leading to a lack of synchronization.

As we can see, dead-reckoning and other types of algorithms will be very inefficient, if not ineffective, in situations such as the above because the state changes are not necessarily predictable and can be of quite short duration. In terms of communications, we can see that whatever communications mechanism is used, it must guaranty correct perception of the latest state of a shared object.

Let us look at another collaborative scenario. This is an example of a "virtual theater", a project at MCRLab which is an effort to create a virtual environment for rehearsals of theatrical plays, as well as performances in front of a large number of remote audiences. In this scenario, actors and directors gather in a virtual room where they rehearse their play. Figure 6 depicts one such environment. The types of collaboration actions in such environment are very similar to those of the example in figure 4. For example during a rehearsal, the director might ask an actress to pick up an object from the floor and put it on the table, then move 2 steps back and say a phrase. In addition, there might be closely coupled tasks such as 2 people lifting the table and putting it at the opposite side of the room.

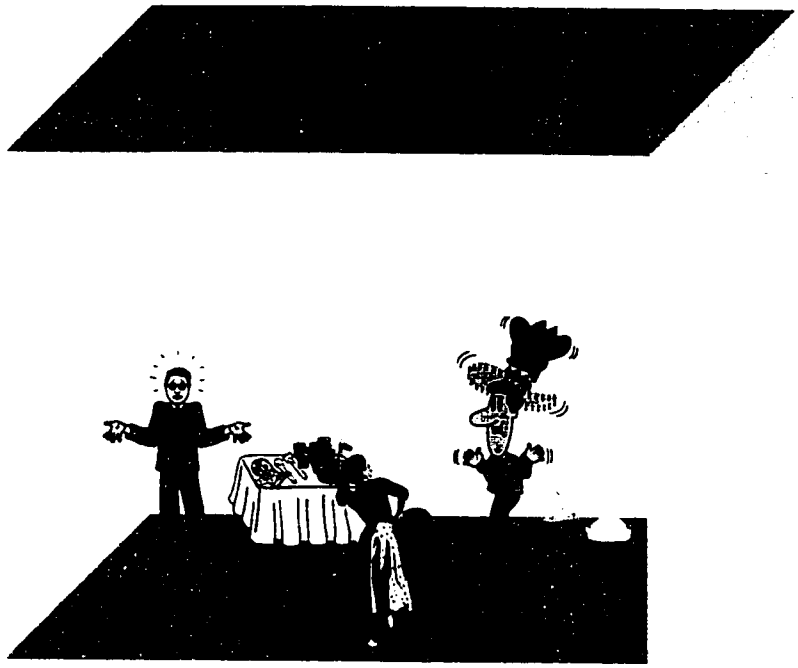


Figure 6. A virtual theater.

All of the observations that were made for the first example are also true in this example. But here we can observe another type of stream: a *long* stream of interaction data, such as the action of lifting a table and carrying it across the room which generates more update messages than pressing a button or rotating a knob. In addition, since there is more than one person involved in interacting with the table, there is a need for all collaborators to see each other's interaction with the shared object and perhaps with one another in real-time. We can then conclude that in such cases there are other interim update messages, other than the last message, which are important and must be received reliably.

Hence, for such tasks that have a long duration or are closely coupled between two or more people, we need assurance that some of the interim update messages in a stream are being received by all participants. It might be acceptable to lose some of the update

messages, but certainly not all. Exactly what percentage and how often these losses are acceptable depends on the actual application.

From the above observations, we can therefore state that Synchronous Collaboration requires **timely** and **reliable** delivery of **"key"** update messages. A *key update message* is an update message that represents the state of a shared object, such that that state does not change for a time equal to or greater than the maximum acceptable skew time.

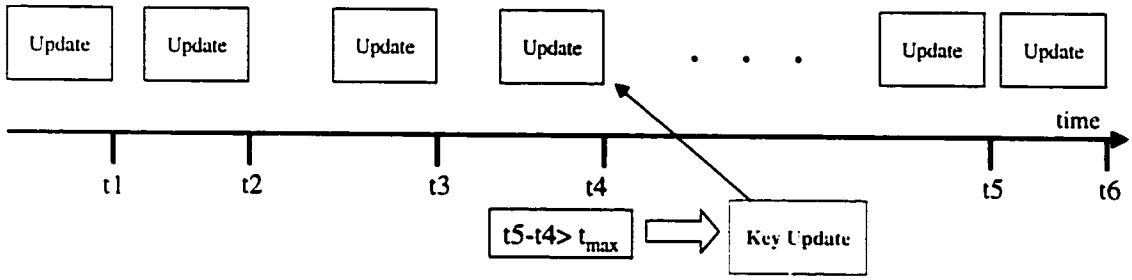


Figure 7. Architectural definition of key updates.

Figure 7 depicts update messages being sent with different time intervals. The update message sent at t_4 is considered a key update since the state of the object remains idle after the transmission of this update message for a duration which is larger than the maximum allowable delay. This overall high-level architecture leads to two lower level architectures discussed next: the Interaction Stream, and the transmission of the Interaction Stream.

4.2 Interaction Stream

Based on the above observations, we can group a sequence of update messages, related to one sequence of interaction of a user with a shared object, into one *stream*. To illustrate this notion, let us have a look at a generic *interaction stream* shown in figure 8.

Figure 8. A generic stream resulting from a user's interaction with a shared object.

This stream will have some critical messages, such as the last update message, which must be sent reliably, and some regular messages which can be sent by best effort transport. Hence, update messages can be grouped into 2 types: *key* update messages, and regular update messages. An interaction stream will then consist of a series of key update messages and regular update messages, as shown in figure 9.

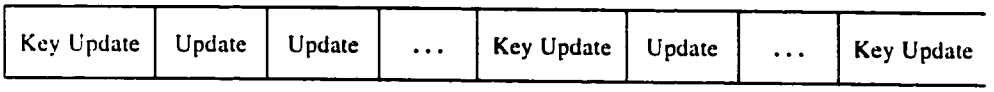


Figure 9. An Interaction Stream.

The last update message will be a key message for reasons explained above. In this scheme, key update messages will be delivered reliably and other update messages in between will be sent by best effort transport.

Determining which updates are key is an application dependent procedure. Generally however, simple approaches can be used where every n^{th} message is sent as key, or every t milliseconds a messages is sent as key, with n and t being parameters that can be configured by the application.

4.2.2 Determining the Last Update Message

There are many ways to determine that an update message is the last one in a stream. One

approach is using the Graphics API. In general, one can determine the last message by monitoring graphics events in the environment in order to intercept when a user releases an object. Today's operating systems and platforms provide the developer with high-level event monitoring capabilities such as when a mouse is clicked, dragged, and released, as well as other user input events such as key strokes. Furthermore, most 3D environments, such as VRML and Java3D, provide an additional level of event interception by putting monitors on objects in order to notify the application when an actual object is touched, moved, and released.

However using the graphics API requires the application to be aware of the architecture presented in this document; i.e., the actual graphical application in execution must be aware that when a shared object is released, it means that the update message generated is the last one in the stream. Though this is certainly achievable in practice, it is also desirable to design the architecture in such a way that a VR application which is not aware of this architecture could also benefit from it. Such design is presented in chapter 6 in the implementation section.

4.2.3 Differential Messages

Another interesting issue is that of "differential messages". A differential message is an update message that instead of carrying information about an entity's current state, carries the difference between the current state and the previous state(s). For example, assume an object moves from location (12245, 156, -1233) to (12243, 155, -1230) and then (12241, 153, -1229). Figure 10 shows two types of interaction streams; one is the regular one and another one uses differential messages.

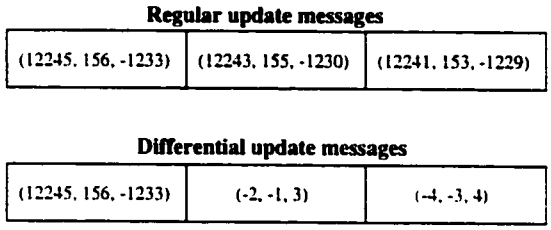


Figure 10. Regular update messages (top) and differential update messages (bottom).

As we can see, differential messages save a lot of bandwidth because instead of transmitting the entire object state, only the difference from the last state(s) is transmitted, which most of the time is smaller than the entire state. Differential messages usually depend on a previously transmitted "reference" update message. In such cases the issue of reliability for those reference updates becomes even more important due to the dependency of many differential updates on one reference update; losing one of those reference updates may results in adverse inconsistencies. Figure 11 shows how differential messages can be sent in the Interaction Stream architecture. The arrows indicate on which reference message a differential message is based on. Reference updates are sent as key messages while regular update messages, which are differential in this case, are sent by best-effort transport. Since the key messages are sent reliably, differential messages will result in the correct sequence of state changes.

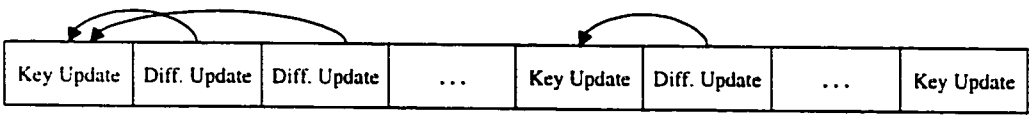


Figure 11. Differential update messages.

This scheme might look familiar to those that are exposed to multimedia encoding standards; more specifically the above scheme looks similar to an MPEG video stream with I, P and B frames. I frames are full frame descriptions which are practically as bulky

as a JPEG image, whereas P and B frames use differential coding based on previous or following frames and are therefore much smaller in size than I frames. From a multimedia communications perspective, it is important that I frames be received reliably because they are the basis for many other frames, whereas the loss of a P or B frame is not as important.

A similar situation applies for CVEs: key update messages must be transmitted reliably because their loss might result in major errors, whereas the loss of a differential message is not as critical.

4.3 Communication

Up to this point we analyzed collaborative update messages and proposed the interaction stream model based on their characteristics. We also mentioned that some updates must be sent reliably while others can be sent using best effort. But how do we actually transmit these messages across the network? Can we use the IP protocol stack and its various protocols for communication?

TCP, UDP, and UDP multicast are considered generic protocols in that they don't make any assumptions about the type of data which they carry. Each of these protocols has been designed for a specific type of service, but not a specific type of data. It is assumed that most data will fall within some category of service requirement, such as reliable, connection-less, best effort, and other types of services.

This assumption does not hold for many types of data. Multimedia communications, for example, requires its own protocols. A good example of this is the Real-Time Transport Protocol (RTP) which was specifically designed for real-time transport of audio and video streams because the above generic protocols alone did not meet the needs of real-time streaming data. Similarly, CVE update messages need their own transport protocol; they require scalability, low latency, and a special type of reliability: timely-reliability. Due to the nature of collaboration, it is pertinent that all parties come into synchronization within the shortest possible time interval. This means that the key updates must not only be delivered reliably, but the reliability has to be timely as well. In the following section, the proposed Synchronous Collaboration Transport Protocol (SCTP) is introduced, which is designed specifically to achieve the above-mentioned tasks, and also takes into account the concept of the Interaction Stream that was introduced in the previous section.

4.3.1 SCTP: The Synchronous Collaboration Transport Protocol

A suitable transport protocol for CVE update messages must meet the following properties:

- minimize generated bandwidth as much as possible in order to be scalable;
- send the update messages as fast as possible to satisfy low latency requirements;
- provide timely reliability for key update messages, including the last message in an

interaction stream.

Note that most collaborative sessions are used among a handful of people, such as a dozen trainees practicing something together, a few engineers designing something together, and so on. But applications where a very large number of people are involved can also exist, such as a huge audience watching a collaboration session. Furthermore, it is possible to have a large number of small groups running on the same network. Due to these reasons, we need scalability.

The first two points suggest some type of a protocol which is based on UDP multicast since it is both fast and bandwidth friendly. This is in fact the reason that so many of the available protocols today are based on multicasting. It's the last requirement where most of the existing protocols either fail or are inefficient. The challenge then becomes to design an architecture that provides timely delivery for all messages, plus timely-reliability only for certain update messages. SCTP, which is described below, has been designed for this purpose.

From an Internet protocol stack perspective, SCTP is a host-to-host layer protocol, comparable to the transport layer in the ISO protocol stack. SCTP is encapsulated into UDP packets and assumes that the underlying physical network supports IP multicasting. In terms of packet format, an SCTP-layer PDU must have enough information in its header to indicate whether it is carrying a regular message, a key message or an acknowledgement. It also must carry the sequence number of the update message in the stream. Figure 12 illustrates an SCTP packet with its fields and their sizes.

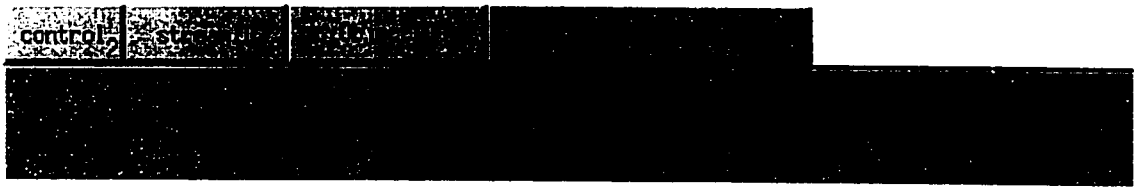


Figure 12. An SCTP packet format.

Let us briefly describe the semantics of each field:

- **control:** 2 bits, indicates the following:
  - 00: regular packet
  - 01: key update
  - 10: ACK packet
  - 11: NACK packet
- **objectID:** 10 bits, contains the ID of the shared object, this is can be set by the VR Environment.
- **streamID:** 8 bits, the ID of the current interaction stream for this object, this field doesn't need to be big since it can be recycled relatively quickly: as soon as the last update message in a stream is acknowledged, the stream number can be used again, although it is recommended to use a different stream number for a while until UDP packets with an extremely high delay (due to possible networking problems) have died.
- **sequenceNUM:** 16 bits, indicates the position of this specific update message in the current stream, there can be 2^{16}=65536 updates per one interaction which should be more

than enough for most interactions. But if this number runs out, SCTP starts a new stream and restarts the sequence count.

- **updateMessage:** arbitrary size (UDP-compliant), this contains the actual update about the shared object (position, orientation, ...). It comes directly from the VR Environment and can be either regular or differential. SCTP simply encapsulates the update info into this field and sends the packet.

Notice that there are no clock or timestamping information in the header. The reason for this is the highly time-sensitive characteristic of collaborative updates: upon receiving an update, it is immediately sent to the higher level where it is implemented at the graphics level as soon as possible. Therefore there is no need for clock or timestamping information. However, since we're assuming a multicast environment, there's always a chance that an older update is received after a newer one which was sent later. That's when the stream number and sequence number information are useful: an update which is older than the newest implemented update, based on the stream and sequence number, will simply be discarded.

Reliability for Key Updates

The next issue is how to achieve reliability for key updates. There are two main approaches for achieving reliability: ACK-based (sender-initiated) and NACK-based (receiver-initiated). Although not in the framework of CVEs, extensive studies have been done regarding these two approaches for Reliable Multicast protocols (RM) and I refer the reader to the literature [7][15][26][29][37][45][50] and refrain from detailed discussion of this topic. In short, ACK-based approaches require the receivers to

explicitly send an Acknowledgment packet to the sender of a packet, so the sender is sure that receivers have received its packet. On the other hand, NACK-based approaches require the receivers to send a Negative Acknowledgment only when they detect a lost packet.

From the above two approaches, NACK-based approaches have gained wider popularity since they have been proven to out-perform ACK-based approaches in terms of network bandwidth and other parameters. NACK-based approaches use Automatic Repeat Request (ARQ) techniques, Forward Error Correction (FEC) techniques, or combinations of both. Most these techniques achieve very high performances for reliable multicast applications, but they are generally either inefficient or unsuitable for the interaction stream scheme presented here.

A host in an ARQ-based scheme only detects a loss when it receives a newer update message with a larger time stamp/sequence number. This creates a problem because in CVEs the transmission of a newer message might happen very late leading to delay, or even worse, never happen at all (nothing is sent after the last update message) leading to collaboration failure. After a user releases a shared object, we don't know how long it takes before the shared object is interacted with again. This "idle time" of the shared object can be any length such as 1 minute, 10 minutes, or 1 day. If the last update message is lost in this scenario, there will be a collaboration failure.

FEC-based techniques group many adjacent packets into a block and generate repair packets based on the contents of the packets in the block and transmit the repair packets at the end of the block. This way if an error occurs in the block, a receiver can repair it

using the repair packets. This technique works quite well for reliable multicast situations, but again it's not suitable for the interaction stream. The problem is that in an interaction stream, adjacent packets don't require reliability. Hence one of the main purposes behind the design of FEC is defeated. There's only one packet in a bunch of a few adjacent ones that requires reliability: the key update. The next key update, which requires reliability, is not produced immediately but only after a few regular updates have already been transmitted. For a sender to collect a number of key update messages, in order to form a block, and then calculate repair packets and send them will cause large delays, and either the early key updates or the repair packets will arrive late at the receivers.

We therefore require a reliability architecture that is ACK-based; i.e., a sender explicitly requests that key update messages be immediately acknowledged. This way a sender automatically re-transmits the key update message in case it is lost. However, ACK-based approaches inherently lead to the infamous "ACK implosion" phenomenon: when every receiver sends an ACK message to every sender's key update, the network could become flooded with ACK messages. Even though in my design only key updates are acknowledged and not all the updates, the fact that there might be a huge number of users in the session creates a potential ACK implosion problem. This problem can be solved using a representative-based technique as suggested by [9][45], such as the following technique.

The first time a key update message is received, all receivers send their acknowledgments. The sender selects the recipient whose acknowledgement is received last as the representative of the group. Next the sender multicasts a control message indicating its choice for a representative. Other recipients will then refrain from sending

ACK messages unless the representative fails to do so. For a given packet, if the representative does not send an acknowledgement, all recipients who have received the packet will send an ACK message based on a suppression algorithm similar to those used for NACK-based approaches, and the sender re-selects a representative from those recipients. The suppression algorithm basically enforces the receiver to time-out for a random amount of time before sending the ACK packet. While waiting, if the receiver receives an identical ACK packet from another receiver, it cancels time-out and does not send its own ACK packet. When a host receives an ACK for an update message that it has not received, it sends a NACK. This NACK can be used for congestion-control purposes. The sender can then utilize the ACK and NACK messages in multicast congestion control mechanisms such as [9] to control its transmission rate.

The above approach uses both ACK and NACK messages to support the timeliness and reliability requirements of updates, but it is mainly ACK-based since the NACK packets are used solely for congestion control. Notice that the above approach is not an actual RM-based protocol; it only use an ACK-based technique for key updates only.

The sender should also keep a record of the representatives IP addresses. When it does not receive acknowledgement from some of the representatives, it re-transmits the key update and specifies precisely which representatives it is expecting an acknowledgement from, so that only those representatives, not all, send back ACK messages. This will further reduce the number of ACK messages.

One question is how to choose the re-transmission timer? Commonly, the re-transmission timer is equal to twice the amount of end-to-end delay, in order to allow the packet to

arrive at the receiver and the receiver's acknowledgment to reach the sender. In our case, as a rule of thumb, the re-transmission timer should be equal to twice the amount of the maximum end-to-end delay between any two participants in the session. It should be noted that for a practical and useful collaboration session, this maximum end-to-end delay should not be more than 200 msec, as described previously. This puts an upperbound of about 400 msec for our re-transmission timer.

4.3.2 Why A New Protocol?

There aren't many protocols that allow one to send reliably only specific packets and unreliably all other packets. One exception is the Selectively Reliable Transport Protocol (SRTP) [31] which was explained in chapter 3. SRTP is a very efficient protocol for generic distributed simulations, but it doesn't quite address the synchronous collaboration issues that were discussed in this thesis, mainly due to its NACK-based approach. Another one is the Interactive Sharing Transfer Protocol (ISTP) [44], but studies have questioned the capabilities of ISTP for tightly-coupled collaborative tasks [23], mainly due to its use of TCP for reliable communication.

One could argue that a hybrid approach, which uses UDP multicast in conjunction with an already available RM protocol, could take care of this problem. In other words, send the regular messages with UDP multicast and send the key messages with an already available RM protocol. However, such hybrid protocols are not suitable for the dissemination of the interaction stream presented here. In addition to the compatibility and implementation problems, the variety of available RM protocols out there do not exactly meet the required specifications dictated by the Interaction Stream, or any generic

CVE update message for that matter. In fact that is the reason that similar architectures, such as the SRTP protocol, use their own custom-made RM protocol instead of using such hybrid designs and taking advantage of the available RM protocols. For example we simply cannot use purely NACK-based RM protocols, either utilizing FEC or ARQ techniques, due to reasons explained in 4.3.1. We can't use ACK-based RM protocols either because in our case we need to drop an old key update which is being replaced by a newer key update, even if the old one has not been received reliably yet. But RM protocols will try to transmit that old packet until it is successfully received by all participants, a feature which is very useful for most RM applications, but is problematic for our approach. Hence the design of SCTP is justified by the lack of other protocol's providing functional requirements for highly-synchronous update messages based on the proposed Interaction Stream.

4.3.3 The Issue of Reliability for Interim Key Updates

It was mentioned in 4.1 that the interaction stream may contain interim key update messages, other than the last key update message (see fig. 8). That was justifiable based on the needs at the application layer.

But in terms of communications, a quick analysis reveals that one does not always need to transport the interim keys reliably. The reason for this is in the fact that the consequent update messages transmitted after an interim key update can be thought of as retransmissions of the latest state. Assume that a key interim update message is lost

which means the sender is not going to receive an ACK message from at least one representative. In this case, the sender is not going to re-send the same interim key update, but it's going to transmit the latest update message, an act which the sender does in any case with regular updates, even if the interim key update was not lost. So in general, there's no need to send reliably the interim key updates. But there are cases that interim key updates must be sent reliably; these are:

- *differential streams.* Key interim updates in the case of streams that contain differential messages must be sent reliably due to the dependency of the differential updates on the reference key updates.

- *when the time interval between successive updates, which is the inverse of the frame rate of the application, is less than the retransmission timeout.* It is obvious that if the framerate is so low that the time interval between successive update messages are larger than the re-transmission timeout, interim key updates must be sent reliably to insure consistency among all participant.

- *for congestion control purposes.* The senders must control their transmission rate to avoid network congestion. In order to do so they need feedback from the receivers. This feedback is in the form of receiving ACK messages and NACK messages, or lack of receiving ACK messages.

This brings us to the issue of congestion control which is discussed next.

4.3.4 Congestion Control

Congestion Control is an important part of any transport protocol, specially those addressing a multicast environment with a large number of users. Basically, congestion control adjusts the transmission rate of the protocol based on the current status of the network and its traffic. When the traffic on the network reaches its bandwidth threshold, the routing nodes start to drop packets and the loss rate increases. If applications continue transmission at their normal transmission rate, heavy losses will occur. In addition, the situation becomes worse for applications using reliable protocols because the senders continue re-sending the lost packets until an acknowledgment is received, adversely contributing to congestion and increasing it. Without congestion control, a collaboration system could suffer significantly when the network starts to lose packets.

Different transport protocols address congestion control differently. In TCP for example, when congestion is detected (by lack of receiving an acknowledgment from a receiver), the transmission threshold is immediately reduced by 50% and the transmission rate itself starts from one segment long and works its way up until it reaches the receiver's maximum window size or until another congestion situation is detected. At that point the procedure is repeated again [1].

While this technique works well for TCP, it is not suitable for CVEs. When the rate is reduced in TCP, the downloading speed goes down and a file which would have taken a certain amount of time to download, will now take more time to download, creating a simple "inconvenience". The same scenario does not hold for CVEs. Update messages must be received within a certain time limit; they cannot be held longer and transmitted

later or at a lower transmission rate. This is because of CVEs' natural rate of data transmission which itself is due to human factors regarding collaboration in the real world.

Instead of reducing the transmission rate, the protocol shall drop some update messages at the sender at the time of congestion. For example, when congestion is detected, SCTP drops every m^{th} update message in a stream but continues to transmit the rest of the update messages at normal transmission rate. This will reduce the overall bandwidth without violating the delay threshold required by the system. Of course the "frame-rate" quality of the session goes down because of fewer update messages per stream, but collaboration will still be possible at its natural rate, up to a certain lower limit obviously. The dropping continues until the congestion situation is over, at which point the dropping rate is reduced until it becomes zero. In addition, because the dropping is done by the transport protocol as opposed to the network, the application has a lot more control over which packets should be lost during congestion, compared to a more random packet loss caused by the network.

There is a great deal of research going on in the field of congestion control for multicast environments. As a matter of fact, this is a huge research field by itself into which I do not intend to be drawn to. In their calculations and mathematical formulas, most of these congestion control mechanisms use a Global Round-Trip Time (GRTT), the computation of which is considered another field of research by itself since the computation of GRTT for a large group is a complex task [2], specially on the Internet, and there's still on-going research on how to do this more efficiently. What I propose to do is to adopt one of these congestion control approaches and use it in SCTP in the near future, when the field is

more stable and some standards have emerged. The advantage is that a proven and efficient congestion control mechanism will be used. As an example of how to achieve this, let us go over an existing congestion control approach.

This approach is a modified version of the work of Dante et al [9]. In their approach, congestion detection is achieved by means of Congestion Clear (CC) and Congestion Indication (CI) messages received from a dynamic set of representatives in the multicast group. A sender adjusts its transmission rate based on mathematical formulas each time a CC or CI has arrived. The authors have further proven that their approach is compatible with TCP traffic, neither suffocating nor completely submitting to TCP traffic, making the approach implementable for the Internet. This approach is beneficial for SCTP because in SCTP we already send ACK and NAK messages; so these messages can be used for congestion control purposes. A modification to this approach, to make it work with SCTP, is to replace the transmission rate (number of bits per second) by the frame rate (number of packets per second) using a simple conversion formula:

$$frame\ rate = (\text{int})\frac{transmission\ rate}{8 \times packet\ size}$$

where packet size is in bytes/packet.

Effect of Congestion Control on Jitter

It was mentioned in the previous chapters that jitter is a very important QoS parameter of CVEs, argued to have an even greater impact than delay. Jitter is caused by many factors. At the networking layer, jitter is caused by network loss leading to retransmission, as well as the dynamics of network traffic leading to different delays at different times.

Controlling jitter is a very difficult task. At a given node in the network, different update messages stay in the queue for different durations at different times. There are many users in the network, each participating in one or more communication sessions. It is very hard, if not impossible, for a transport layer protocol to control how long a packet spends time in the network.

For multimedia applications, the problem of jitter is usually addressed by a "delayed buffering" mechanism; i.e., incoming packets are buffered at the receiver, delayed for a given amount of time, and played back smoothly and at a constant rate. This technique works quite nicely for *presentational* applications such as video on demand. But for *conversational* applications, where more than one person interacts with the environment in real time, the buffering delay is a major weak point and something that must be avoided. Due to the strict latency requirements of CVEs, buffering of the update messages at the receiver is not an acceptable solution.

At the transport layer, one of the things that can be done to prevent jitter is to try to keep the overall network traffic below the capacity of the underlying network. In other words, by reducing the chance of packet loss and by keeping the queue size of the nodes in a network relatively constant, the system can improve the jitter problem to some extent, although it does not eliminate it. Using a congestion control mechanism allows the transport protocol to react to the network traffic in such a way that the message rate (and hence the traffic generated by a source) is decreased when the network traffic increases, and it goes up when the network traffic decreases. This means that congestion control should theoretically decrease the chance of packet loss and also keep the overall traffic of the network relatively constant, leading to an improved jitter characteristic.

4.3.5 SCTP Traffic Generation

An important question is how much traffic do the ACK packets in SCTP generate in addition to the normal traffic generated by the application? This question is important in estimation of the extra traffic generated in a collaboration session in order to demand and allocate QoS parameters such as bandwidth.

To answer this question, we first notice that ACK packets are sent for each key update, at the end of the stream, or for each re-transmitted key, which occurs at every retransmission time-out. Let the re-transmission timeout be t_{retx}, we can then divide the time axis from the moment a key update is received into equal intervals with length t_{retx}. Let us call these intervals re-transmission cycles, or simply cycles. At each cycle, a number of ACK packets are generated by the representatives. Initially, the number of packets is equal to the number of representatives as each representative must send an ACK. In the second cycle, depending on how many ACK packets were lost on their way to the sender, some or all representatives might have to send again ACK packets in response to the sender's retransmission. Similarly in the third cycle there might be a need for some more ACK retransmission, and so on.

Analyzing this trend, we can see that the number of ACK packets generated in a given cycle should be less than, or equal to in the worst case, to the number of ACKs generated in the previous cycle since the chance of losing all ACKs from the previous cycle decreases with increasing number of cycles. The following is a more in-depth analysis of the generated traffic.

Average Traffic

Let us make the following definitions:

N= number of representatives,

p= probability of packet loss across the network

At any given time, the Binomial Theorem tells us that the probability of losing x packets out of y becomes:

P(losing x packets out of y)=

$$p(x,y)=\binom{y}{x}p^{x}(1-p)^{y-x} \qquad (1)$$

where $0 \le x \le y$.

Now, in a given cycle x, let us represent the chance of losing y packets by $P_{x,y}$. In order to lose y packets in a given cycle, we must have lost at least y packets in the previous cycle. $P_{x,y}$ can therefore be calcualted as follows:

$P_{x,y}$=P(y or more packets were lost in cycle x-1 AND y packets are lost in cycle x)

= P(y pkts lost in cycle x-1).p(y,y) + P(y+1 pkts lost in cycle x-1).p(y,y+1)+...

+P(N pkts lost in cycle x-1).p(y,N)

Hence:

$$P_{x,y} = \sum_{m=y}^{N} P_{x-1,m} \cdot p(y,m) \qquad (2)$$

where $1<x<\infty$, and $0 \le y \le N$.

For the above series, we define the following initial condition:

$P_{1,y}=p(y, N)$,where $0 \le y \le N$

This is intuitive since at x=1 we know that exactly N packets were sent previously and the chance of losing y packets is simply p(y,N).

We can now define a "Loss Matrix" P, which consists of elements $P_{x,y}$ as defined above. Each $P_{x,y}$ indicates the chance of losing y packets in cycle x. This Matrix has a dimension of c×N+1 (since there can be from 0 to N packets lost in each cycle, so N+1), where c is the number of cycles.

For example, assume a packet loss rate of 15% in a session with 3 representatives. The loss matrix over 5 cycles is shown below. Notice how each row adds to 1, due to the law of total probability.

$$P = \begin{matrix} 0.6141 & 0.3251 & 0.0574 & 0.0034 \\ 0.9340 & 0.6455 & 0.0015 & 0.0000 \\ 0.9899 & 0.0101 & 0.0000 & 0.0000 \\ 0.9985 & 0.0015 & 0.0000 & 0.0000 \\ 0.9998 & 0.0002 & 0.0000 & 0.0000 \end{matrix}$$

Now, NP_c, which shall denote the average number of ACK packets generated in a given cycle c can be computed as:

NPc=P(losing zero packets in cycle c)×0 + P(losing 1 packet in cycle c)×1 + ...

+ P(lossing N packets in cycle c)×N

Hence:

$$NP_c = \sum_{j=0}^{N} j \bullet P_{c,j} \qquad (3)$$

;where c>1, and NP_1=N.

Therefore, the total average traffic generated from the beginning until cycle c is the sum of NP_c components and can be computed as:

$T_c=NP_1 + NP_2 + \ldots + NP_c$

$$\therefore T_c = N + \sum_{i=1}^{c} \sum_{j=0}^{N} j \bullet P_{c,j} \qquad (4)$$

where c>0, and T_0=N initially.

Figure 13 below shows a plot of T_c for N=30 and retransmission time of 400 msec, and for various loss rates over 5 retransmission cycles.

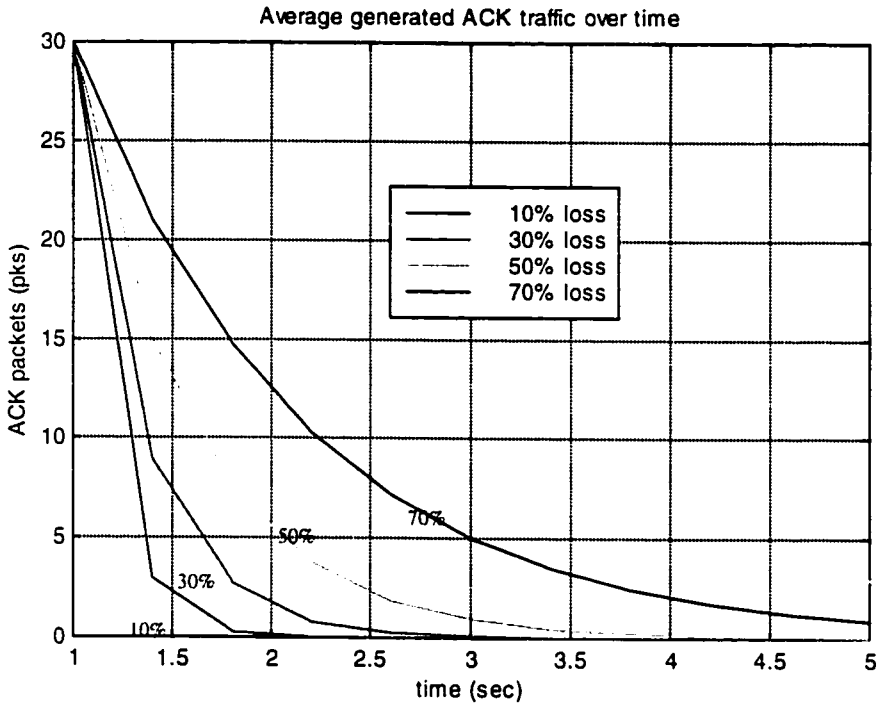


Figure 13. Average traffic generation over time.

As we can see, the average generated traffic decreases substantially over each cycle. In fact when analyzed by MATLAB, it was revealed that the traffic decreases geometrically in such a way that $NP_{c+1}=p.NP_c$ for $c\geq 1$; where p is the network packet loss described at the beginning of this analysis. This decrease is consistent with the theory presented earlier, since it was mentioned that the chance of losing more packets decreases with increasing number of cycles, so the traffic must also decrease with increasing number of cycles.

The graph also shows that even for unpractically high and abnormal loss rates, such as 70%, the generated traffic quickly converges towards zero.

Since the terms $\{NP_1, NP_2, \ldots, NP_c\}$ form a geometric series with the multiplier p, T_c can

be thought of as the sum of a geometric series. We can therefore find the total average traffic resulting from a key update as:

$$T_{total}=\lim_{c\to\infty} T_c \qquad (4a)$$

but p<1; hence:

$$T_{total} = \frac{N}{1-p} \qquad (5)$$

By looking at equation 5, it is clear that there is a linear relationship between the total average traffic generated by a key update and the number of representatives in the session. The graph of figure 14 shows the total average traffic generated due to ACK messages in terms of number of representatives. We can see that this traffic changes linearly.

Notice that figure 14 does not plot the traffic versus the actual population of the collaboration session, but versus the number of representatives, which is a fraction of the total number of users in the session. Therefore the traffic versus actual population has an even better performance than that shown in figure 14. For example, at a representative rate of 10%, Number of Representaives=20 in figure 14 actually refers to a population of 200 participants.

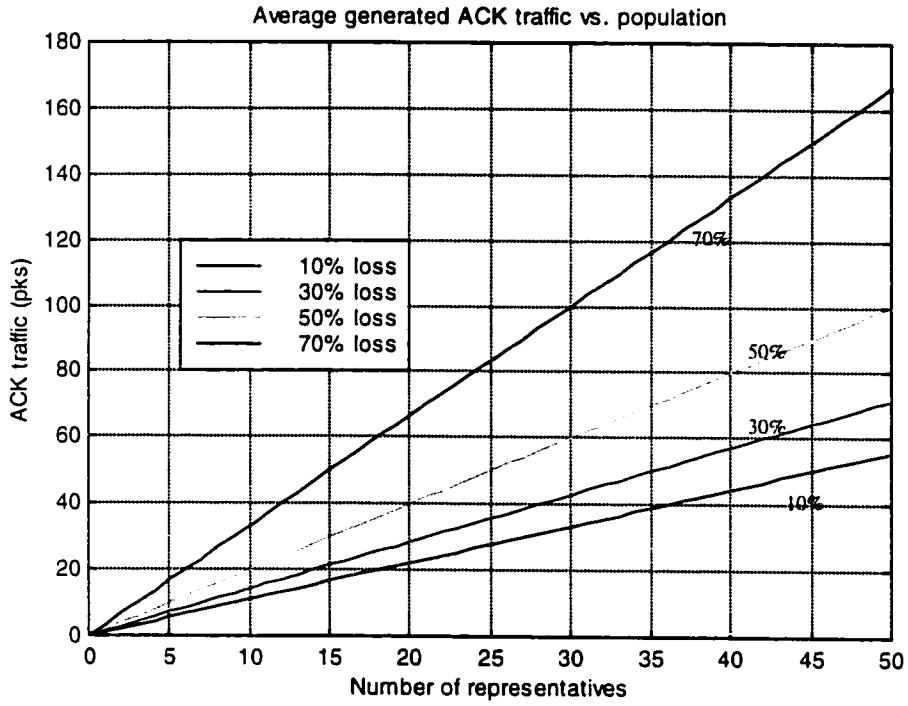


Figure 14. Average traffic generation vs. population.

Exceptions

One of the assumptions made in the above analysis was that all representatives receive the key update in the first cycle and send ACKs. But what if some of them don't receive the key update in the first cycle, or consequent cycles? Since packet loss for each representative is independent of others, the average traffic generated will be unaffected by whether or not the first message is lost by a representative. Hence, on the average we will still get the same amount of traffic.

Also, the above analysis does not take into account interim key updates. Although a similar situation applies to the key interim messages, there's always a chance that the next interim key update be received while the ACK traffic from the previous one has not quite

approached zero. In such case the total traffic for each key update is simply based on the time interval between interim key updates; i.e., in equation 4a the parameter c is finite. In fact, figure 13 can be used to decide when to transmit an interim update message as key, in such a way as to make sure that the resulting acknowledgement traffic has approached its limit on the average. Remember tough, that interim key updates are transmitted reliably only in specific situations (see 4.3.3).

Maximum Traffic

All of the above analysis was for the case of average traffic. There is a need to also find the maximum traffic. Maximum traffic is generated in the worst case scenario, which occurs when all representatives send ACKs, but all ACKs are continually lost over each retransmission cycle, forcing all representatives to re-send ACKs in each cycle. Though the chance of such sequence is extremely low for all practical purposes, it must be considered in this theory section as a worst-case scenario. This means that N ACK packets are sent at each cycle. Figure 15 on the next page shows the generated ACK traffic over time in this case.

This means that for a retransmission timeout of t_{retx}, we are going to have, in the worst case, an additional N/t_{retx} pkts/sec traffic for each shared object, which again changes linearly with respect to the number of shared objects. However, as mentioned earlier the chance of such scenario is extremely remote. In fact as we can see from figure 16 on the next page, this chance approaches zero very quickly.

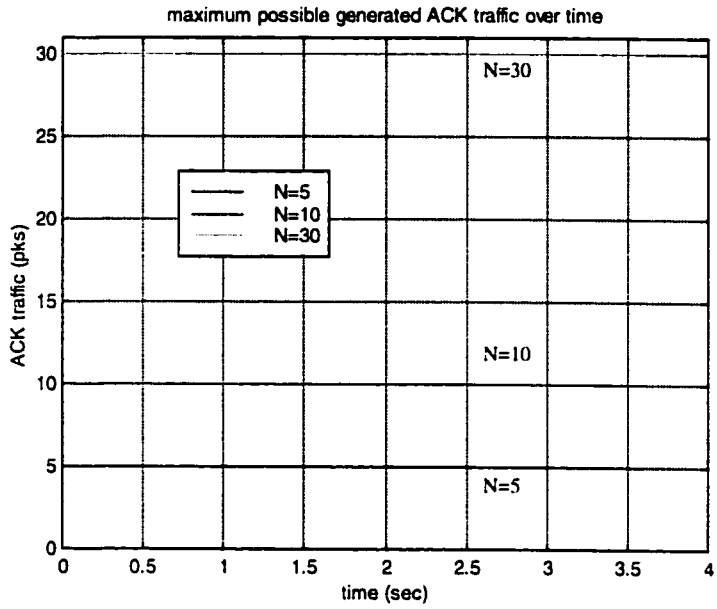


Figure 15. Maximum traffic generation (t_{retx}=400 msec).

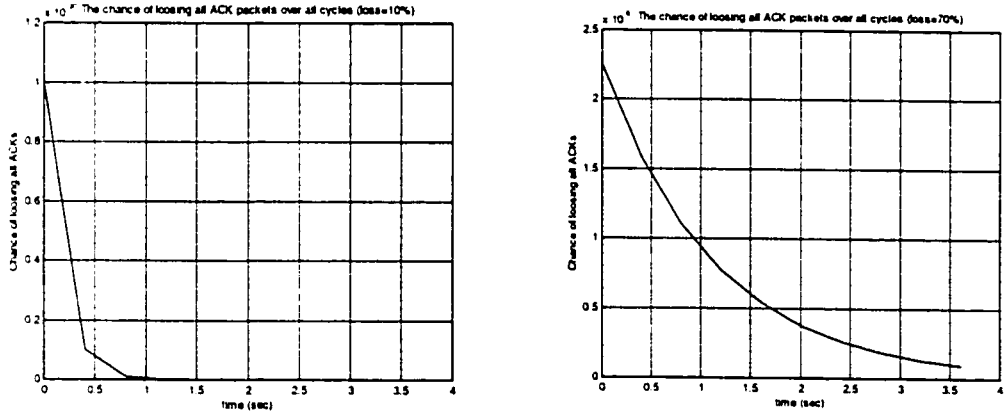


Figure 16. The probability of losing all ACKs in all cycles.

4.4 Comparison with Similar Work

Two novel and key concepts were shown in this chapter: the Interaction Stream application model, and the SCTP communication protocol. SCTP was in fact designed

based on the Interaction Stream model and is fully dependent on it.

No other work has addressed the issue of tightly-synchronized collaboration in virtual environments in this way. The work by K. Park et al does focus on collaboration in CVEs [27][28], but only to the extent of the effect of jitter and delay in collaboration. It does not go into any detail about collaboration itself, and proposes neither an application model nor a transport protocol to achieve more efficient collaboration in large environments. In terms of communications alone, there are two other research works that take similar approaches at the transport layer, but they do so without any design consideration for collaboration data. G. Kessler et al use the concept that the last state of an entity is the most important data in a CVE [12] in order to design a communications protocol. Their technique, called the Inter Process Queue (IPQ) uses the following communications scheme based on UDP messages:

Each sender has a buffer where it stores the last update message, as well as control messages. Each update messages is acknowledged by a receiver. A sender then re-transmits an update message until an acknowledgement is received or when a newer update messages is generated. This newer update message replaces the old one in the buffer and the same procedure is repeated.

Initially, this scheme was designed to run between two people and the authors claim that the same approach can be extended to a many-to-many communications topology. However, it is apparent that in a many-to-many scenario the UDP messaging translates into UDP multicast, if efficiency is to be maintained, and together with the suggested ACK-based approach this architecture becomes very similar to an ACK-based reliable

multicast architecture, with the exception that not every update message is retransmitted: only the last one representing the last state is retransmitted. However, ACK messages are still sent for each update message, leading to the ACK explosion problem.

Another work which is similar to SCTP in terms of its transmission mechanisms is the SRTP protocol. As described in 3.1.9, the SRTP protocol supports 3 modes of operation. Mode 0 is unreliable multicast, Mode 1 is a NACK-based reliable multicast with a NACK suppression algorithm in order to avoid NACK implosions, and Mode 2 uses either TCP or ACK-based multicast between two entities only. Recently a "heartbeat" message transmitted infrequently on mode 0 has been proposed [31] to help receivers decide whether they have missed a message. The SRTP protocol is a very efficient transport protocol for regular update messages in distributed simulations. It does not however support the requirements of collaboration as discussed in this thesis. As criticized earlier, an ARQ-based NACK approach is not suitable for a CVE because of the delay associated with the detection of a lost message, especially the last message. Although not quite the same, the proposed heartbeats are similar in nature to DIS's keep-alive messages, consuming additional bandwidth and still not guaranteeing timely delivery, for collaborative purposes, of an object's last state because a receiver must first receive a heartbeat message to realize it has lost a message, then it must send a NACK after a given suppression time-out, and then wait until it receives the missing packet. This creates a larger delay than when a sender simply re-sends a packet upon time-out. In addition, the heartbeat messages are introduced as a concept only and there is no specification as to how many, at what frequency, and for how long they should be transmitted. So, there is a need for an ACK-based approach which seems to be provided by mode 2 of SRTP, but

this mode is envisioned to send messages reliably between two entities only, leading to a peer-to-peer and therefore not scalable architecture.

4.5 Overall Analysis

We cannot really compare the proposed Interaction Stream to anything similar, mainly because there is nothing similar to it in today's CVEs. This Interaction Stream represents the collaborative actions of a user/entity with a shared object, since it closely mirrors how collaboration is achieved among humans in the real world.

The SCTP protocol, which is based on the Interaction Stream concept, seems to appropriate for dissemination of update messages. Its only disadvantage seems to be the additional traffic generated due to ACK messages. However, timely-reliable delivery of key updates is the most important issue in tightly-synchronized collaboration, and the additional traffic generated by SCTP while trying to support this timely-reliability is the price we pay for synchronous collaboration.

The important point regarding this additional traffic is for it not to be unpredictable or unstable in nature. The theoretical analysis of section 4.3.5 showed that this extra traffic on the average changes linearly with respect to the number of representatives, or in the worst case linearly with respect to the number of shared objects. Which means that with a given number of shared objects and representatives, the extra traffic is constant and the total traffic increases linearly over time. So theoretically the protocol should be stable in

terms of traffic generation.

It seems that on paper, the above concepts all make sense and should work; however proof of concept is required. In the next two chapters, we step from the world of concept and theory into the world of reality and application. The simulation and implementation results are presented in chapter 5 and 6 respectively, which act as proof of concept to the above claims.

Chapter 5. Evaluation

After presenting the theoretical justification behind the proposed research presented in chapter 4, it is time to see how well all of these concepts work in the real world. Based on the context of the proposed protocol, one must prove that the architecture is in fact more efficient and more suitable for collaborative update messages in CVEs than existing approaches. Since the presented architecture is based on the collaboration in the real world, one must prove that it works for actual "collaboration" scenarios where other architectures either fail or are less efficient. To achieve this objective, both simulation and implementation of the architecture are created. In both cases, SCTP is compared with some other protocols, but which protocols? The protocol to compare SCTP with must be carefully selected. In order to do that and before proceeding to simulation, let us do an objective evaluation.

5.1 Objective Evaluation

One cannot exhaustively compare all transport protocols with SCTP. Not only that's time-consuming and unpractical, but also scientifically it doesn't achieve much because it is quite apparent that some protocols do not even qualify to be compared with SCTP. For example, it is quite obvious that protocols such as UDP or RTP, which have no reliability mechanism, will give results that in terms of collaboration are far worst than SCTP. In the

case of UDP, if the last update message of an object is lost, that object will be out-of-synch for as long as a new update message is transmitted. This duration is undetermined and can be short, or very long. It therefore follows that SCTP will easily out-perform an unreliable protocol such as UDP.

At the other extreme, a fully reliable protocol such as Reliable Multicast protocols will have a very high bandwidth requirement and higher delay characteristics compared to SCTP. RMs ensure reliability for all packets, even those that are obsolete in terms of collaboration, creating additional and unnecessary network bandwidth.

A good protocol to compare against SCTP is therefore something similar to SRTP. SRTP provides reliability only for selected updates, hence reducing bandwidth. In the simulations one can make a comparison between SCTP and an SRTP-based protocol in terms of bandwidth and OST. With this in mind, we now proceed to simulation.

5.2 Simulation

The OPNET modeling and simulation tool [68] was used to study the behavior of SCTP. SCTP was fully modeled, except for the dynamic representative feature. A static multicast network was created with 54 hosts and a 200 msec diameter. 10 fixed hosts were strategically chosen as fixed representatives. 20 shared objects were randomly placed, each on one host. Shared objects were set to randomly produce update messages, at 10 frames per second, based on an "idle time" indicating for how long an object is idle before producing the next interaction stream., and a "stream length" indicating the number of update messages per stream. The idle time was uniformly distributed between

0 and a maximum adjustable upperbound, the stream length followed a Rayleigh distribution centered around 20 packets. Rayleigh distribution was chosen to make the occurrence of very short streams as unlikely as very long streams. Based on the above traffic, the router buffer sizes were experimentally adjusted in such a way as to produce a given network loss rate. The experiment was done for idle times with upperbound of 5 and 10 seconds, and network loss rates of 15% and 30%. Notice that such loss rates are high for today's networking infrastructure, but they are common for multicast environments [36]. In addition, a "non-SCTP" protocol, which was based on the SRTP, was modeled as well. The reason this protocol is called "non-SCTP" and not SRTP is that SRTP was not fully modeled. Specifically, mode 2 and the heartbeat message were not used in the model. Mode 2 was not modeled because it is not used by the Interaction Stream, and the heartbeats were not modeled because as mentioned in 4.4 they are conceptual with no specifications determined yet. Retransmission timer for both protocols was set at 400 msec.

In addition, the following components were developed for the simulation:

- **router** node with run-time adjustable *service rate*, *routing table*, and *buffer size*;
- **interaction stream generator** with run-time adjustable *frame rate* (packs/sec), *object idle time* (uniformly distributed), and *average packets per stream* (+ve Gaussian distribution);
- generic **user host** emulating a participant with capabilities such as being a *sender*, *receiver*, and *representative*, all adjustable during run-time;

- probes to measure the required statistics.

The OPNET network, node, and process models are shown in the three figures shown on the next two pages. The network consists of three levels: backbone, gateway, and router level, with each router hosting a host, each host symbolic of a LAN environment. Each level is a sub-network of its parent level (figure 17).

The node, which represents a participant station, consists of processes to generate update messages, receive update messages, and communicate update messages based on the protocol (SCTP or non-SCTP), as well as hardware transceivers (figure 18).

The SCTP and the non-SCTP communications protocol were described in OPNET's ANSI-C-based finite state machine (FSM) process model (figure 19). Both of the communications protocols have the same states, though the ACK state of SCTP is replaced by the NACK state of non-SCTP. However, the interior code of each state is quite different between the two. Both the SCTP and the non-SCTP processes were programmed to ignore the interim key update messages. As explained in chapter 4, these interim key updates only require reliable transmission in specific cases, none of which applied during the simulation.

The interaction stream generator was also described by an FSM (figure 20). This works in the following way: first, a stream length based on the random distribution of the *stream length* parameter is produced. A number of random update messages, equal to the stream length produced, are generated. Then, key updates are chosen based on the *framerate*. Next, updates are transmitted based on the framerate; i.e., after sending one update, the process sleeps for an amount of time equal to the inverse of framerate, and the sends the

nest update. Once the stream is sent, the process sleeps based on a random *idle time*.

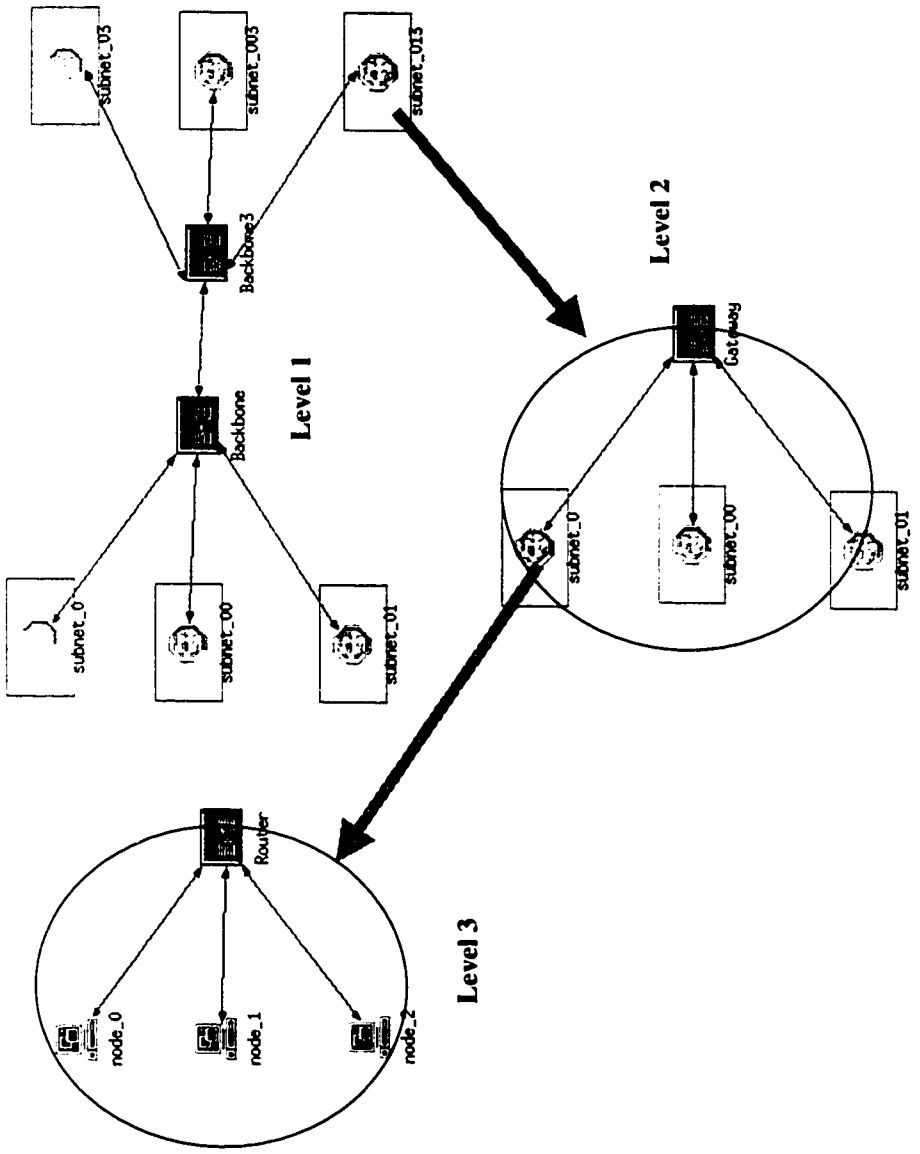


Figure 17. Static multicast network configuration for the simulations.

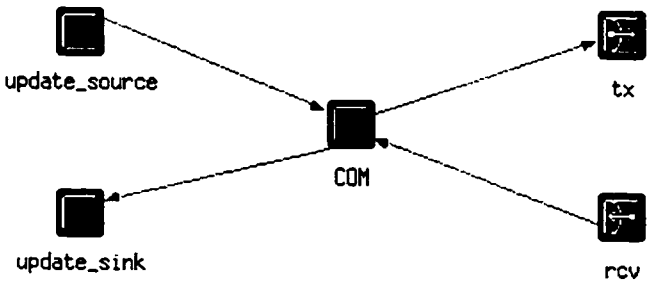


Figure 18. Node configuration for a participant station.

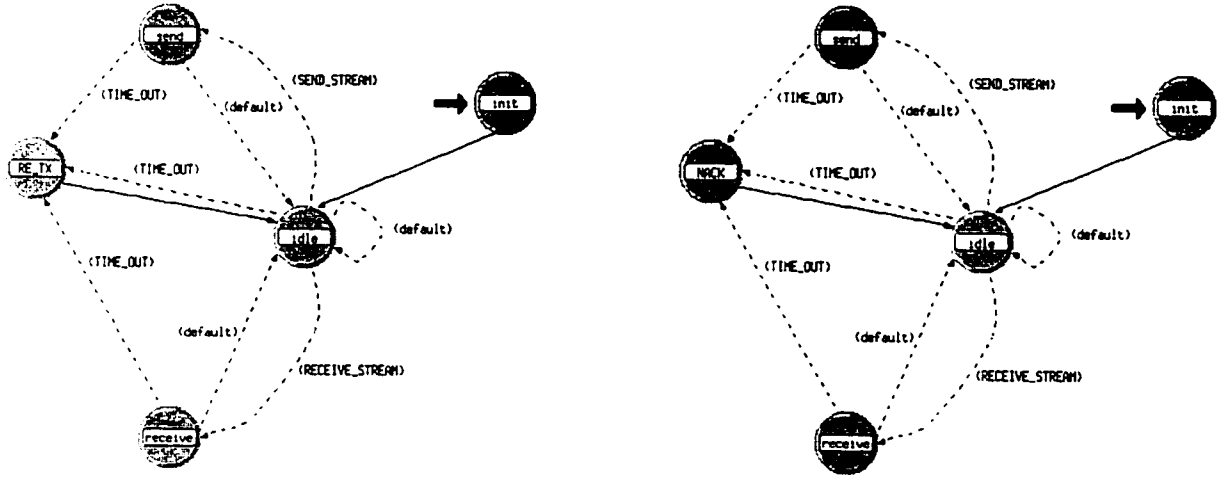


Figure 19. The COM protocols' finite state machine (left SCTP, right non-SCTP).

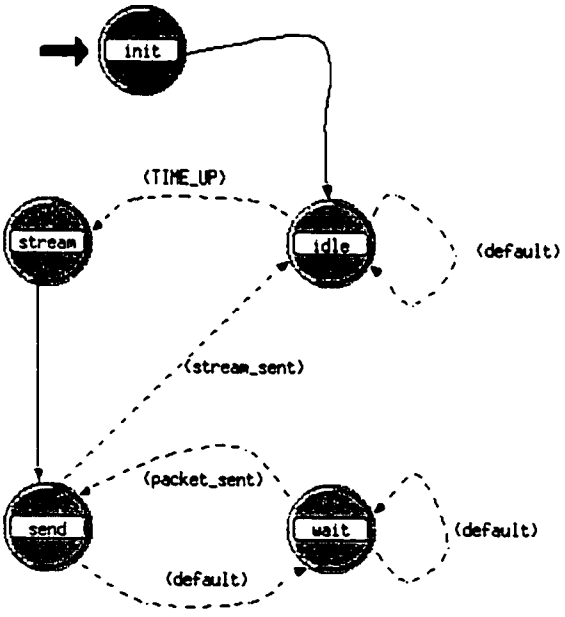


Figure 20. The interaction stream generator.

5.2.1 Simulation Results

The simulations were done for the worst case scenario; i.e., each host on the network is interested in every shared object. This is worst case because the we must send update messages from each shared object to every host on the network, causing maximum traffic which can lead to maximum packet loss. In the simulations, we are particularly interested in "collaboration failures", measured by Out-of-Synch-Time and defined as:

> "**OST** (Out-of-Synch-Time): is defined as the amount of time from when the representation of a shared object on a receiver's machine becomes different from its original object due to a lost message, until it gets back into synch by receiving the last state of the shared object".

OST goes beyond delay, jitter, packet loss, and the usual network simulation parameters: it focuses specifically on collaboration. During OST, the host is said to be in the *Out-of-Synch State*, or OSS. A host would exit the OSS state as soon as it receives a newer update message (in case of non-SCTP) or the re-transmission of the last one (in case of SCTP). Using the data from the simulations, MATLAB was used to parse through the OPNET simulation results and plot the OST values. The results are shown in figures 21 to 23. Each figure presents the SCTP performance results at the top, and the non-SCTP results at the bottom. It also shows the OST values as both average and maximum amount of time, for each host (horizontal axis).

Each graph also shows an Inter-Sequence Delay (ISD). This value is defined as the

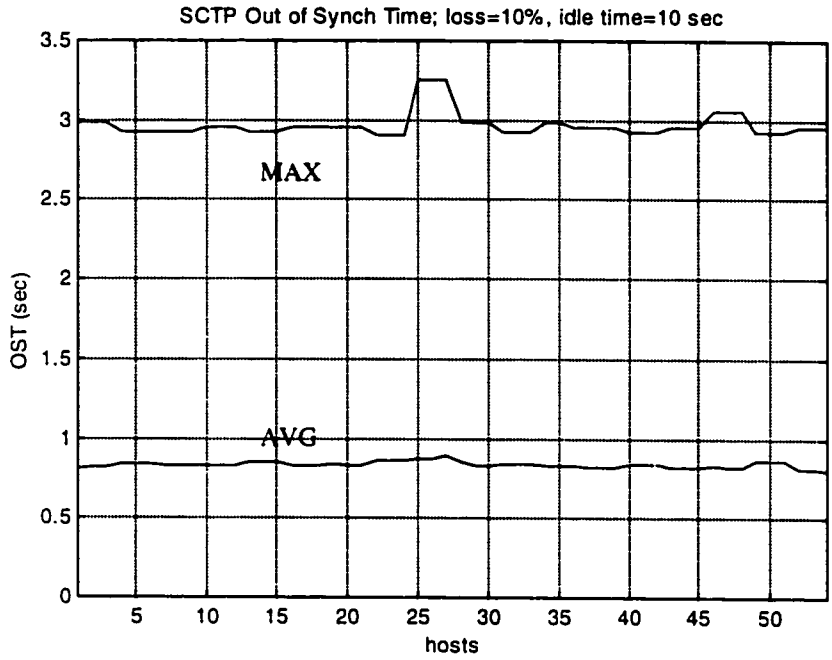
average time between the first time an interim update message is lost until the next one arrives. This parameter was measured to validate my claim for ignoring interim key update messages by the communications protocol.

OST measures for how long an object has a different state on a given host compared to the object's actual state. The higher the OST, the less efficient collaboration will be. In addition, we can introduce "catastrophic failures" as when the OST is larger then a given threshold, as defined by the application. For example, if an object is out-of-synch for more than 5 seconds, than we have a major collaboration problem.

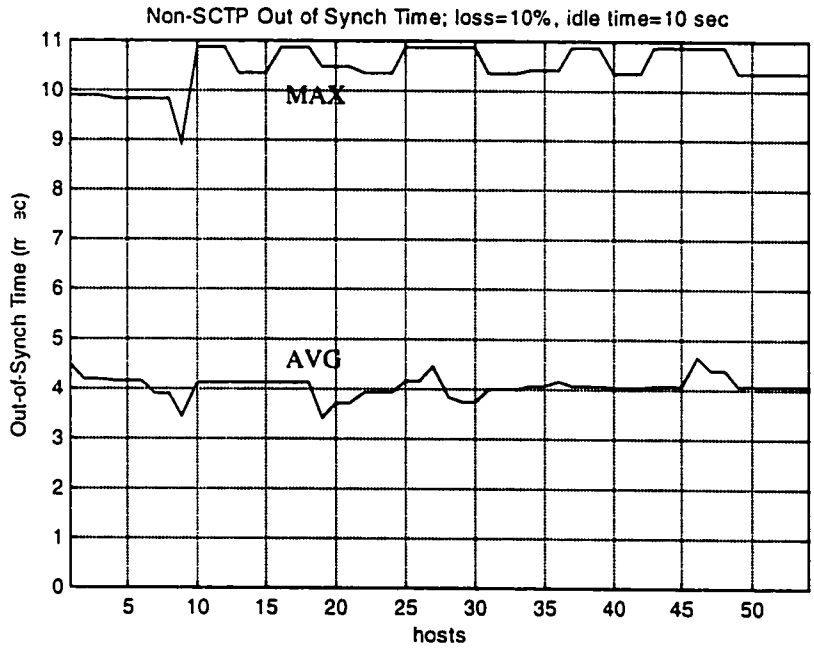
To calculate the OST values, MATLAB was programmed to use the following procedure: For each sender, MATLAB looked at the streams that were sent. For each stream, the sequence number for the last update message is recorded. Next, for that specific sender, stream ID, and sequence number, all receivers are parsed to find out if they all received the key update. If no, the receiver is in an OSS state and the difference between the last update message the receiver received and the next update message are calculated as OST. This procedure is repeated for all sender, all streams, and all receivers, not just the representatives.

To calculate the ISD values, MATLAB simply looked at all streams received and found locations where the sequence numbers of two consecutive updates in the same stream differ by more than 1.

In addition, the total generated traffic was also measured. The traffic results are shown in figure 24. All of these results are analyzed in the next section.

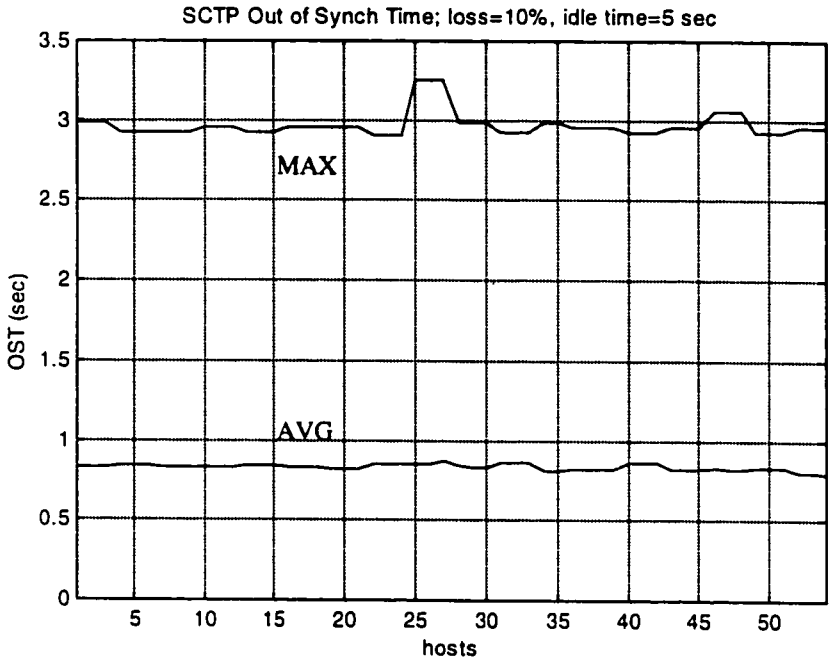


ISD=250 msec

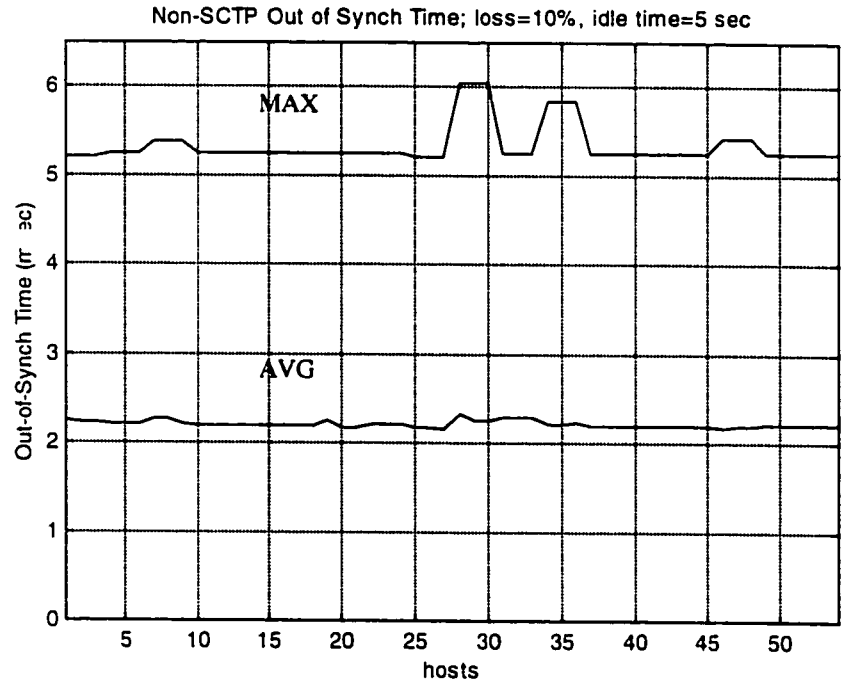


ISD=261 msec

Figure 21. OST values for loss=10% and idle time =10 sec.

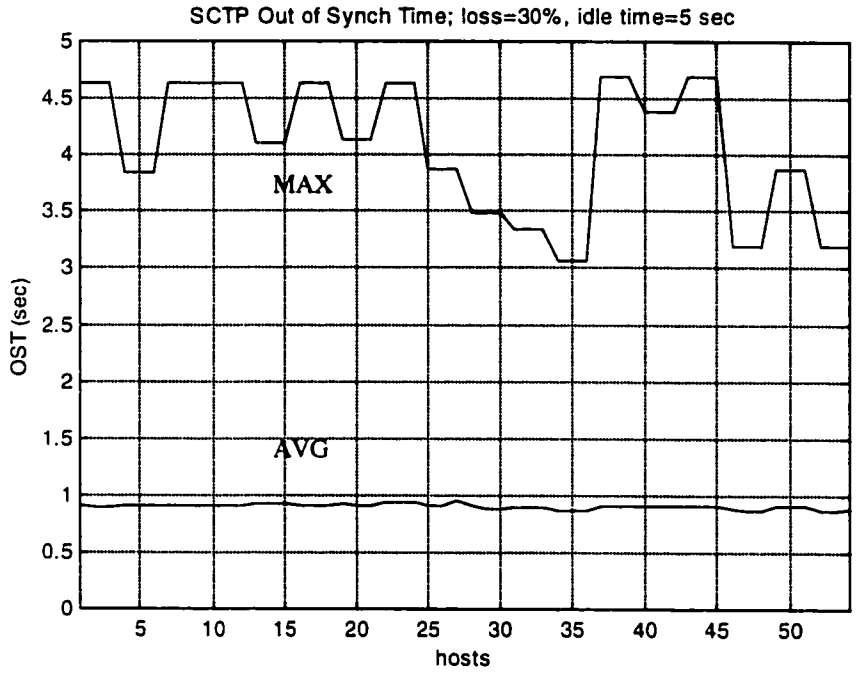


ISD=241 msec

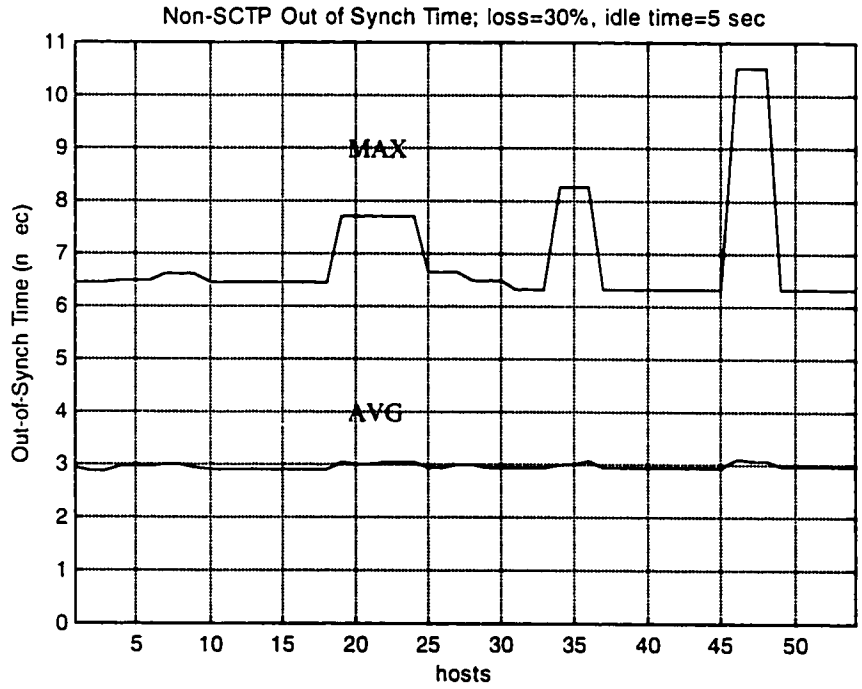


ISD=268 msec

Figure 22. OST values for loss=10% and idle time =5 sec.



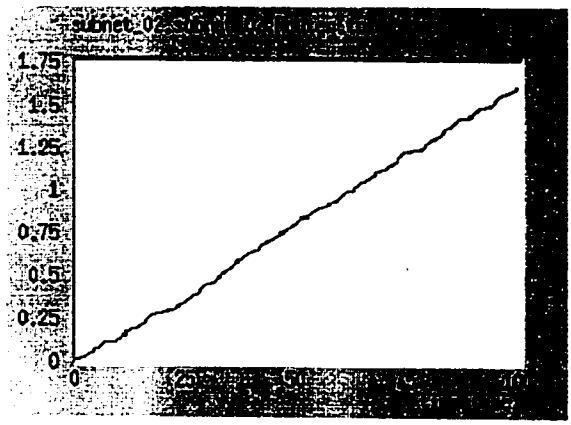
ISD=324 msec



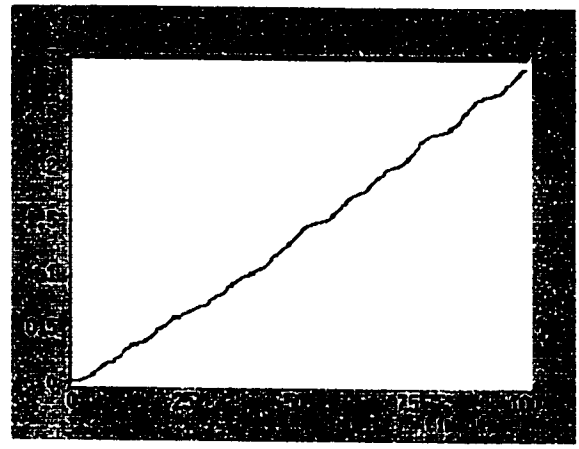
ISD=423 msec

Figure 23. OST values for loss=30% and idle time =5 sec.

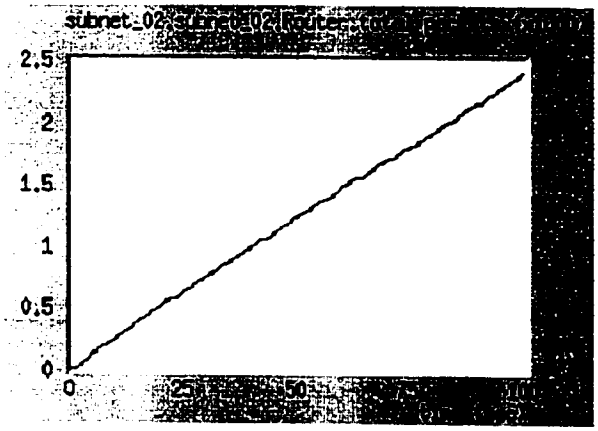
Traffic (non-SCTP loss=10%, idle time=10 sec)



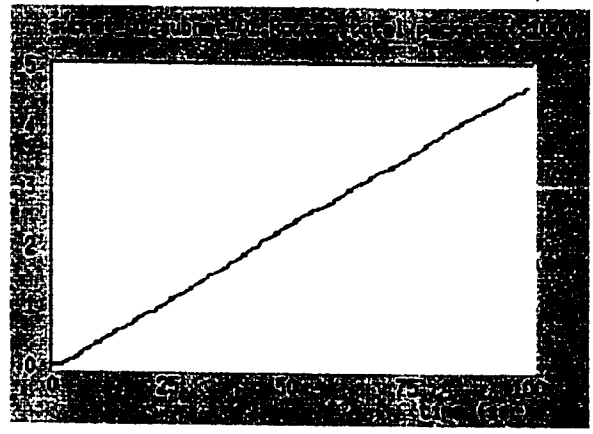
Traffic (SCTP loss=10%, idle time=10 sec)



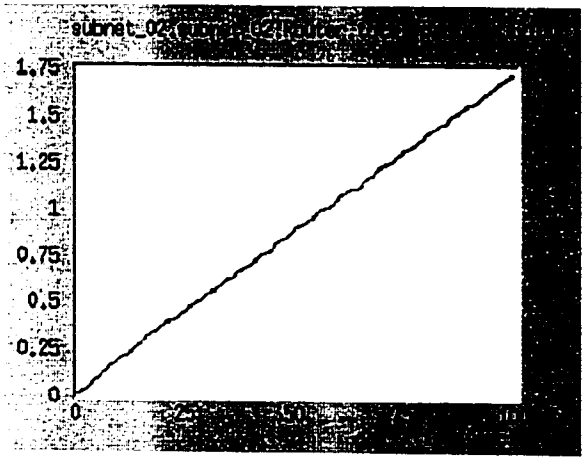
Traffic (non-SCTP loss=10%, idle time=5 sec)



Traffic (SCTP loss=10%, idle time=5 sec)



Traffic (non-SCTP loss=30%, idle time=10 sec)



Traffic (SCTP loss=30%, idle time=10 sec)

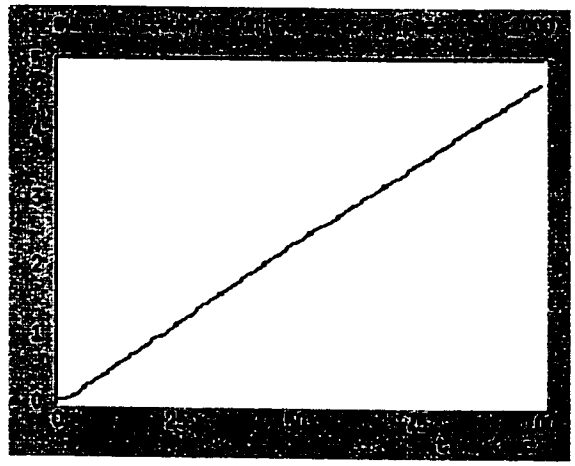


Figure 24. Total network traffic vs. time.

5.2.2 Analysis

Note that the graphs only show the OST for the last update message. As mentioned earlier, under normal circumstances there is no need to send the interim key updates reliably. This claim is substantiated by the obtained ISD values in figures 21 to 23. As we can see, for both SCTP and non-SCTP protocols, the average ISD values are considerably smaller than the retransmission timeout. It's only during higher loss rates that the ISD values approach the value of the retransmission timeout, but even in these cases they are still much smaller than the OST values. This means that lost interim key updates are replaced by newer updates at a speed which is better than what the re-transmission mechanism can do.

Now, comparing only the SCTP graphs of figures 21 to 23 (the top graph in each figure), we can observe the following:

1- The average OST value does not change significantly with increasing loss rate; it is always around 750 to 800 msec.

2- The average and maximum OST value does not change significantly with increasing object idle time.

These results are intuitive due to the timely-reliable delivery mechanism of the SCTP architecture. SCTP keeps retransmitting the last state of a shared object until it is sure all representatives have received it. Therefore the average OST value is affected significantly neither by the loss rate nor by the idle time of an object. In fact, the main effort in this thesis was to create an architecture that behaves in this manner, in order to

efficiently carry out collaboration in CVEs. These results validate that effort.

Since the retransmission timeout was set at 400 msec, the smallest possible OST value for SCTP is 400 msec. The largest one depends on how many "cycles" were required to complete a reliable transmission for a key update, and it differs from case to case. Hence the obtained 750 to 800 msec time for the average OST is consistent with the theory.

Comparing only the non-SCTP graphs of figures 21-23, we can observe that the maximum and average values of OST are heavily dependant on both the loss rate and the idle time. In fact in the case of loss rate=10% (figures 21 and 22), we can see that the distribution of OST is almost identical to the distribution of the object idle time; i.e., uniform distribution from 0 to maximum idle time. This is no coincidence. As explained previously, when the last update message is lost, an object can potentially be out of synch until the first update of the next stream arrives: a duration equal to the amount of time the object is not sending updates and hence is idle. This means that the average duration of each failure is very close to average amount of time an object is idle (offset by network delays). Similarly the maximum OST is close to the maximum idle time. Furthermore, the OST value increases to even higher values with increasing loss rate, because at higher loss rates the chance of losing the first few packets of the next stream increases also, hence adding to the amount of time the host is out of synch.

In all cases, comparing OST values between SCTP and non-SCTP graphs shows that the SCTP protocol easily out-performs the non-SCTP protocol. Again, the main reason for this is the timely-reliable characteristic of SCTP which is not provided in non-SCTP protocols.

In terms of traffic, figure 24 clearly demonstrates that the total traffic in the network for SCTP increases linearly. Again this is consistent with what was expected based on the theory of section 4.3.5, and shows that the protocol is stable. Also, we can see that compared to the non-SCTP protocol, the traffic generated by SCTP is larger. For our settings with 20 shared objects and 10 representatives for 54 hosts (≈20% representative population), the SCTP traffic in terms of number of packets is about 2 times more than the non-SCTP traffic. This is the price we pay for the ability to perform synchronous collaboration, which was also expected and explained in section 4.5. Note that all packets are not identical though: ACK packets are 36 bits in size compared to the 116 bits of the update packets.

Finally, for the sake of completeness, the actual delay and loss rates for different simulation cases are presented in figures 25 and 26.

The next chapter presents the implementation of the proposed architecture.

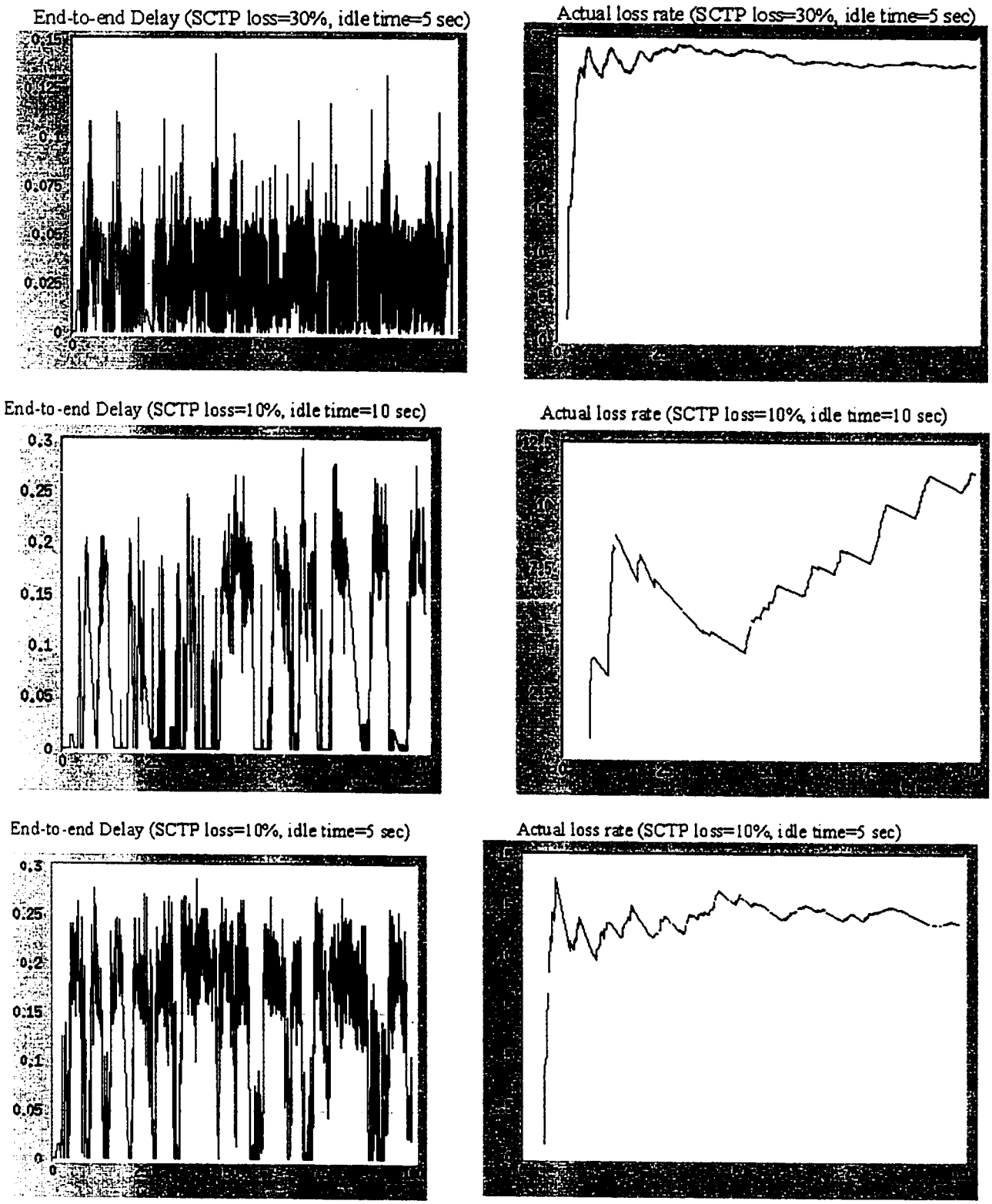


Figure 25. Delay and Loss results for SCTP.

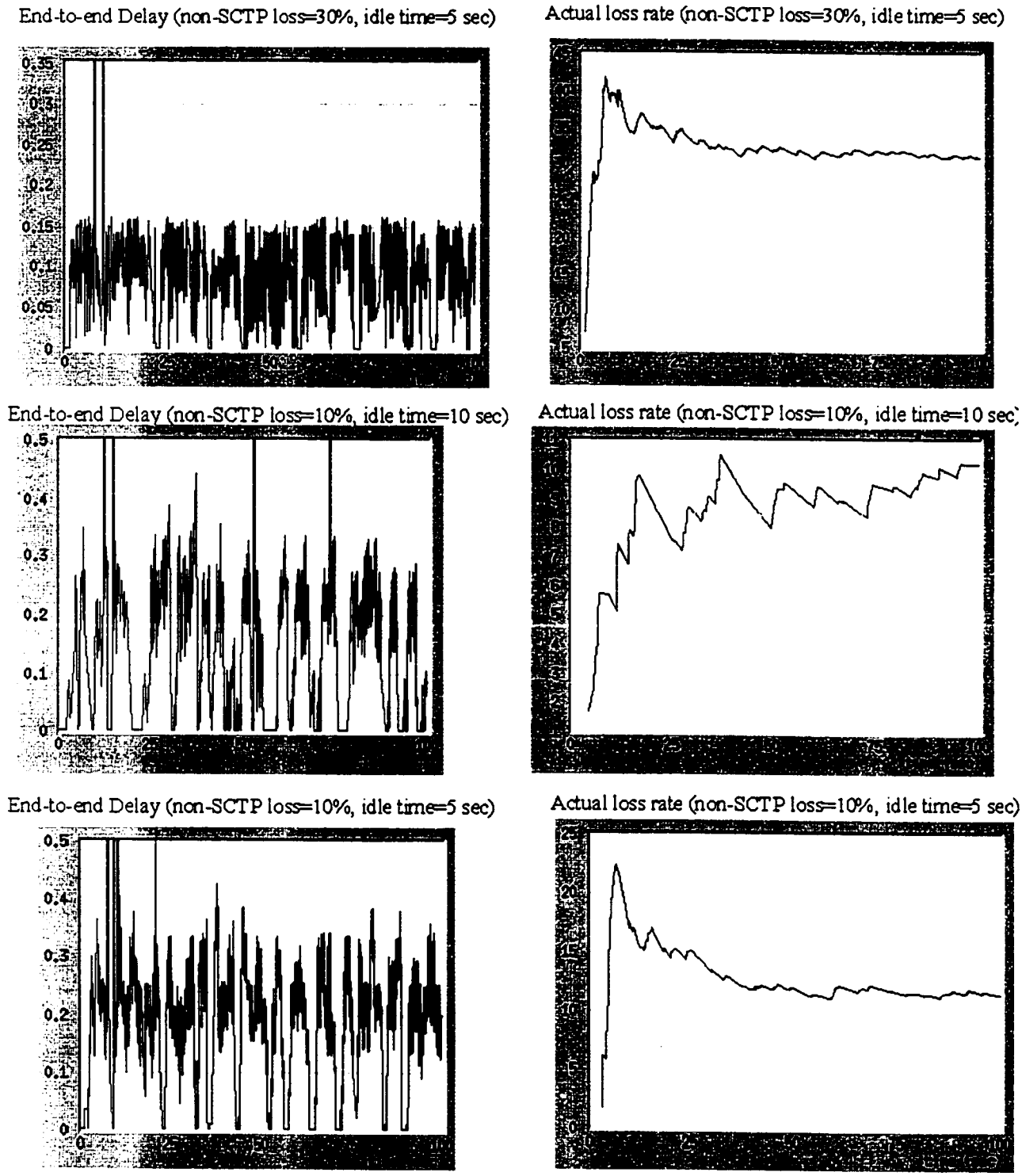


Figure 26. Delay and Loss results for non-SCTP.

Chapter 6. Implementation

The architecture presented in chapter 4 was implemented in order to observe its performance in the real world and with real human participants in actual collaboration sessions. In this chapter we will see the implementation details, the performance tests, and the results.

6.1 Functional Requirements

Two main things had to be implemented:

1- The architecture presented in chapter 4, which itself consists of two parts:

- Interaction Stream;
- SCTP.

2- A test application that is suitable for testing the implemented architecture.

The Java language was used to write the code for the implementation. The reason for this was mainly ease of use, and the ability to seamlessly integrate the architecture into other MCRLab projects which mostly use Java and Java3D. The performance issue of Java is not very critical in this case, because at the MCRLab we had found out that Java adds

about 200 microseconds of additional delay on top of the native networking API[1]; therefore, since for networked applications the delays generated on the Internet are much higher, the use of Java is not a bottleneck in such applications. In terms of Graphics, Java3D wraps around native OpenGL libraries and for not-too-complex 3D applications it has an acceptable performance when used with graphical accelerator cards.

6.2 *The INVENTIST Framework*

The architecture presented in chapter 4 was implemented as a middleware framework. This framework, called INVENTIST for INception of Virtual ENvironments' TIghtly Synchronized Tasks, can be used either stand-alone, or a middleware plug-in to be used in higher-level architectures such as CAVERNSoft [20], HLA [18], and The Virtual Reality Transfer Protocol (VRTP) [10], as shown in figure 27.

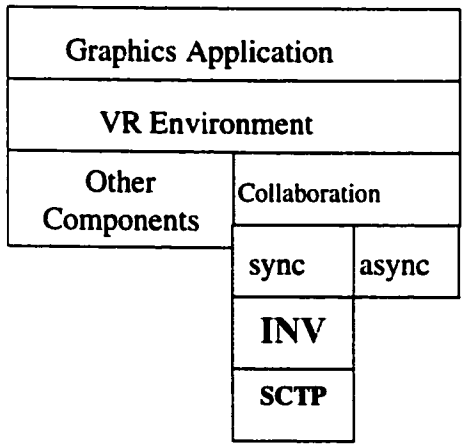


Figure 27. INVENTIST as a plug-in middleware.

[1] Tested on a Pentium II 333 Mhz machine with 128 Meg RAM.

There reason for using INVENTIST as a plug-in is that the proposed architecture only focuses on synchronous collaboration, and not other aspects of virtual environments such as object registration, access control, floor control and moderation, database issues, and other services. The architectures cited above already address these issues and it is logical to provide synchronous services to those existing systems since many good environments have already been developed and are in use. As we will see, the INVENTIST middleware includes an "adapter" that can be used by higher-level systems and applications to connect to it and utilize its synchronous collaboration services. For the remainder of this section, the Unified Modeling Language (UML) representation is used to explain the implementation of INVENTIST. Please note that to reduce the size of this document, only very high-level details about the system will be presented. Details and in-depth demonstration of the system will therefore not be shown in this thesis. For details, the readers are referred to the Application Programming Interface shown in the Appendix. Developers can furthermore use the comments I've put in the source code to understand details of the implementation.

Figure 28 shows the overall design of INVENTIST. As far as the VR application is concerned, INVENTIST provides a simple mechanism to send and receive update messages. It does so by using the Interaction Stream module which itself utilizes the communications module. Notice the interface separation between the Interaction Stream module and the communication module. This allows any communications module to be "plugged" into the INVENTIST system. For this implementation the proposed SCTP protocol was used. In the future, if a different protocol needs to be written, due to an innovative networking technology for example, one can simply replace SCTP by that

newer protocol without changing the rest of the INVENTIST framework. That newer protocol must preserve the integrity of the interaction stream and also provide timely and reliable transmission for key update messages.

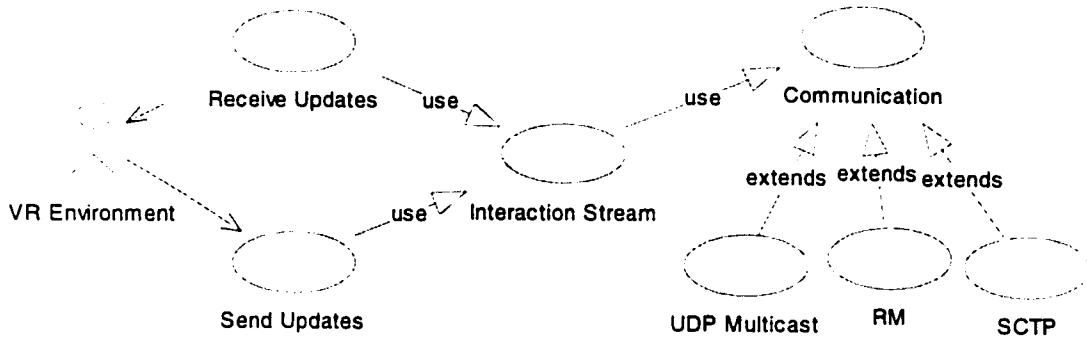


Figure 28. INVENTIST Use Case Maps.

Figure 29 shows the implemented classes. Again, as far as the higher level VR application is concerned, INVENTIST consists of simply an interface, the Inventist interface, which the VR environment can implement (InventistImpl) to use the synchronous collaboration services provided by INVENTIST such as adding shared objects to the system, sending updates, manually adjusting the frequency at which key updates are sent, automating the key generation process, and more.

As we can see the communications module is also an Interface (Comm Interface), which can be implemented by any developer. The implementation of it (CommImpl) must implement the necessary methods such as sending and receiving updates, and joining and leaving multiuser groups. In my case the Comm interface was implemented by SCTP. In addition, the framework has some internal classes and utilities to take care of the concept of stream, shared objects, timers, and so on.

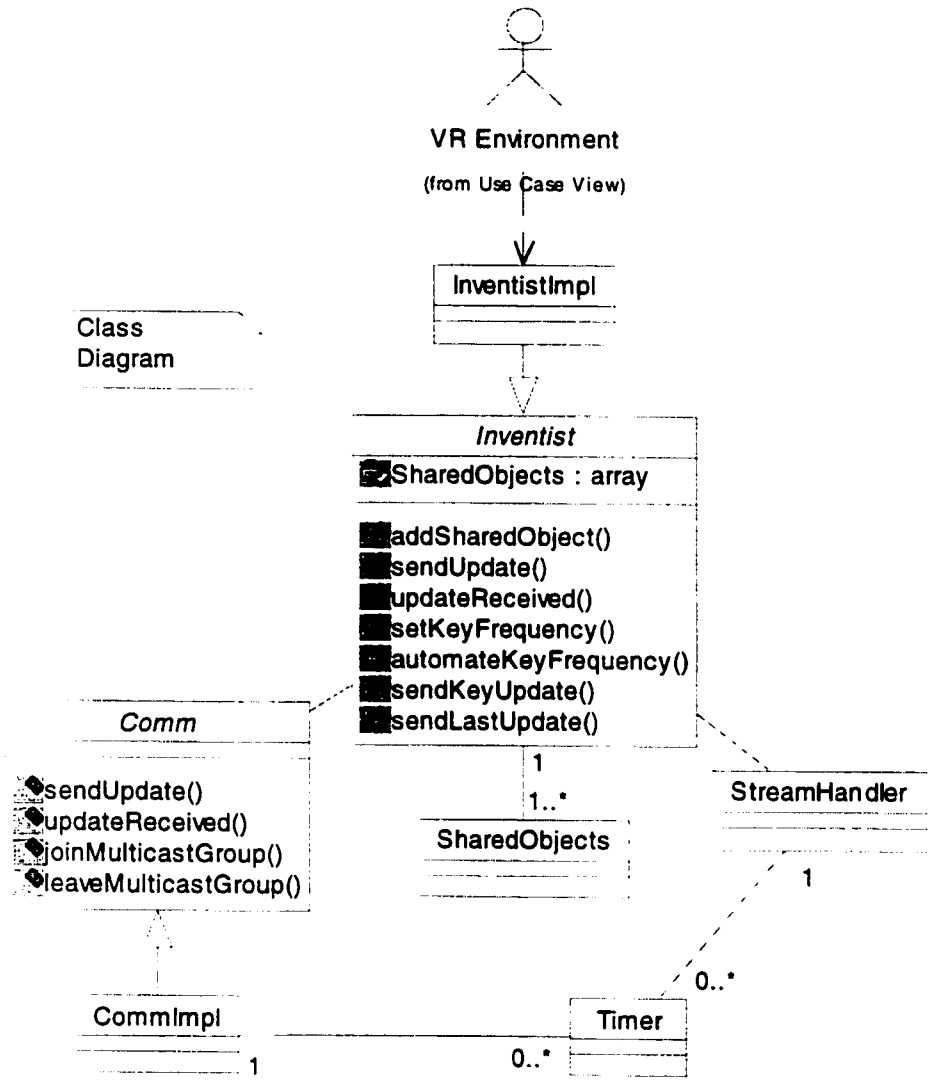


Figure 29. INVENTIST Class Diagram.

Figure 30 is a UML sequence diagram that shows how INVENTIST automatically determines the last update message. As mentioned before, the ability to determine the last update in a stream is an important one because most applications will be unaware of the concepts discussed in this thesis. These applications are not necessarily collaboration-aware, and expect that by using INVENTIST the issue of synchronous collaboration is

addressed and they don't have to worry about how it's done. This is a reasonable expectation. Therefore INVENTIST provides a built-in feature to determine the last update message.

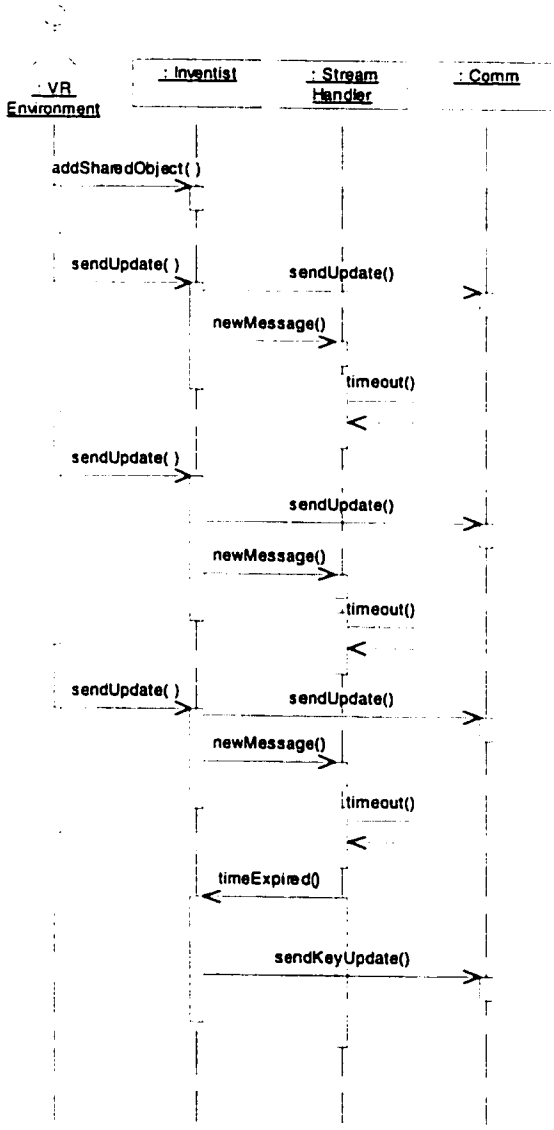


Figure 30. Automatic key estimation by INVENTIST.

As we can see in figure 30, this is done by a time-out functionality in the StreamHandler

object. Every time INVENTIST receives an update form the VR application, it sends this update both to the communication module and to the StreamHandler object. The streamhandler object immediately goes on a 200 msec timer. If another update is received during this time-out, the timer is cancelled and a new one is re-started. If the timer times out, it means that no other update message was received and the last update message sent is considered to be the last one in the logical stream and is re-sent as key. 200 msec is chosen in order to respect the latency requirements of CVEs.

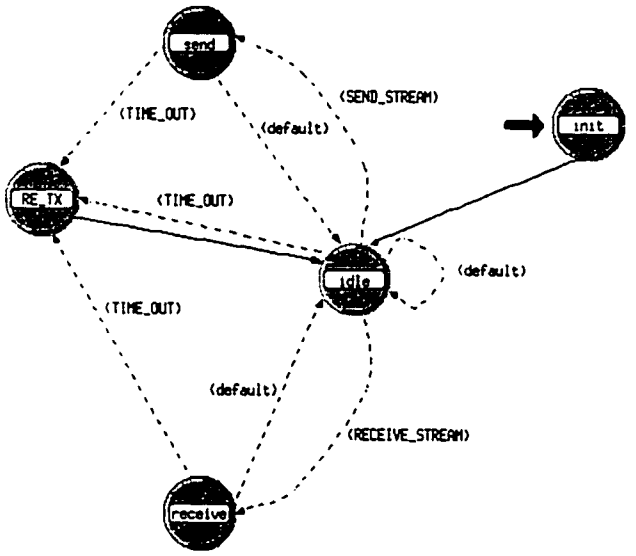


Figure 31. The SCTP finite state machine.

Figure 31 shows the finite state machine of the SCTP implementation, which is identical to that of the simulation. After initialization, SCTP can have 3 non-idle states: sending a packet, receiving a packet, and re-transmitting a packet. When a packet is received from the network, SCTP first examines the packet's header fields to see if an acknowledgment is required, in which case it send an ACK response to the group. Otherwise, in the case of

a regular update message, it checks to see if the message is valid based on its sequence number and stream ID. If the update is late it simply discards it; otherwise it passes it to the Inventist class which immediately passes the update to the VR application. The sequence of these actions are shown in figure 32.

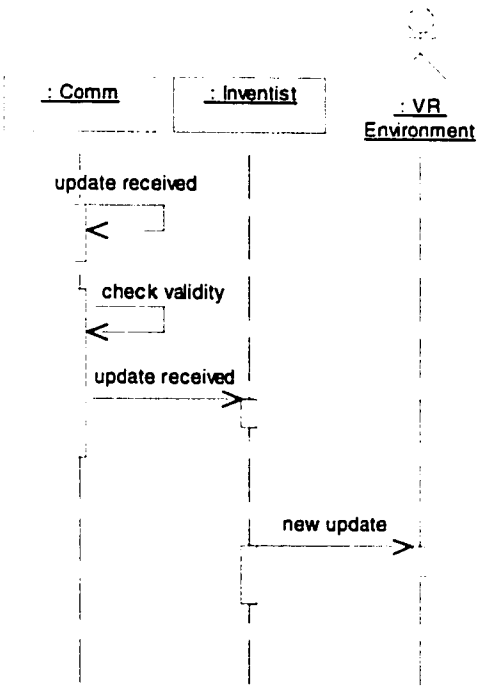


Figure 32. passing an incoming update from network to the application.

The "send" state of SCTP occurs when an update arrives from the Inventist class. Based on the StreamHandler's processing, the Inventist class notifies SCTP whether to send this update as key or not, and whether it's the last update in a stream. The sequence numbering is performed at this stage as well. Based on the above info, SCTP fills in the header fields of the packet, encapsulates the update message itself in the packet, and sends the packet to the underlying UDP multicast.

For each key update, SCTP starts a timer. If the ACK messages from all representatives arrive before the timer expires, SCTP cancels the timer. Otherwise, the timer times out and forces a re-transmission state. While in that state, SCTP re-transmits the last key update message and also re-starts another timer.

In addition, the non-SCTP protocol described in chapter 5 was also created as an implementation of the Comm interface.

6.3 Classes and Interfaces

The classes and interfaces of INVENTIST were packaged based on their functionality into packages shown in figure 33.

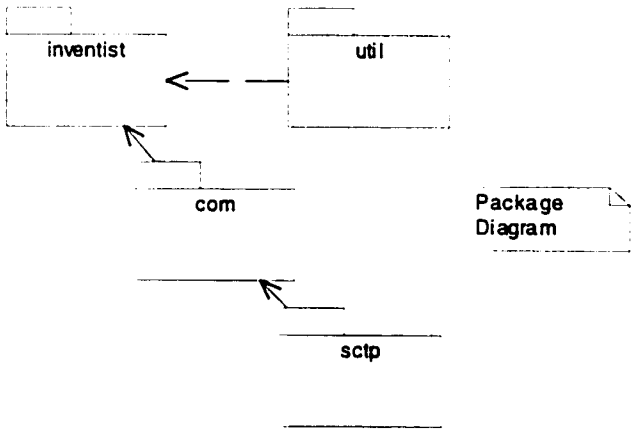


Figure 33. Package Diagram

The position of each class/interface in a package is shown in figure 34. Notice that figure 34 also includes the classes for testing, and the surgery application which we will see in the next section. The complete list of the methods and descriptions for each class, or the

Application Programming Interface (API), is presented in the appendix at the end of this thesis. In general, all classes are configurable based on the parameters discussed before, such as n, t, whether the process of finding the last key should be done automatically or not, re-transmission timeout, last message time out, and other parameters. The INVENTIST framework was then used to develop a simple collaborative application. This application, which is a tele-surgery simulation, is presented in the next section.

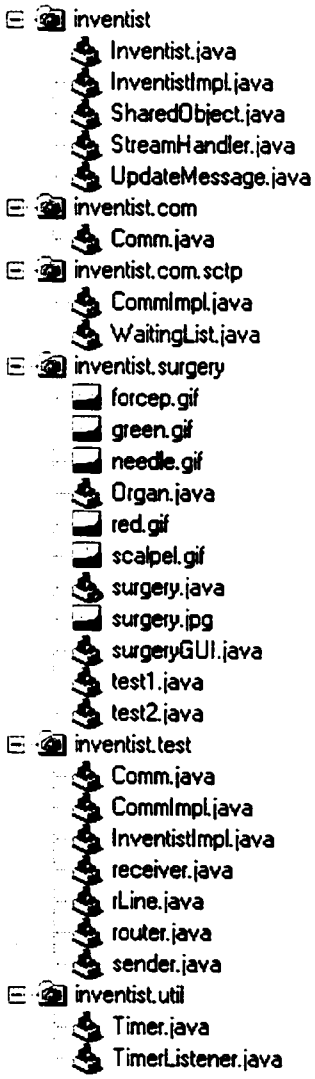


Figure 34. INVENTIST Packages and classes.

6.4 The Tele-Surgery Application

Surgery is an operation that requires very tightly-coupled collaboration between members of the surgery team. In fact the lack of collaboration can lead to fatal "errors". So it was decided to create a simple tele-surgery application, where 2 or more surgeons can operate on a patient. The idea was to have at least two surgeons: one using a forceps to move gradually a long body organ, such as an intestine, out of the patient's body while the other surgeon uses a scalpel to cut specific parts of the organ. To make the task more precise, special "hot spots" were identified on the organ, so that cutting and stitching was only allowed on those hot spots. These hot spots were identified by a black color while the rest of the organ was light red. Between each two hotspots is an organ "segment". The first surgeon pulls out the organ segment by segment, each time waiting for the second surgeon to cut the hot spot. After all hot spots were cut, the first surgeon gradually pushes the organ back into the body segment by segment, each time waiting for the second surgeon to stitch the hot spot. Figure 35 shows a screen shot of the application.

Once cut, a hot spot turns red. If stitched, it turns white. This was done to assist the participants know about the status of hot spots. In addition, tele-pointers were used to create a sense of awareness between the surgeons: each surgeon could see the movement of the tool of the other surgeon which not only would indicate where the other surgeon is pointing to, but also what tool he/she is using. This can be seen is figure 35 on the next page, where it is clear that surgeon 1 is using a forceps to hold the organ, and surgeon 2 is using a scalpel and is in the process of cutting a hot spot. It also shows that the second and third hot spot from the top have been cut and stitched, respectively. One can also see

other features such as the patient status indicator, which turns red if something goes wrong, and the ability to change, on the fly, the test parameters such as the application frame-rate and communications protocol. If SCTP is not used, the non-SCTP protocol is used automatically.

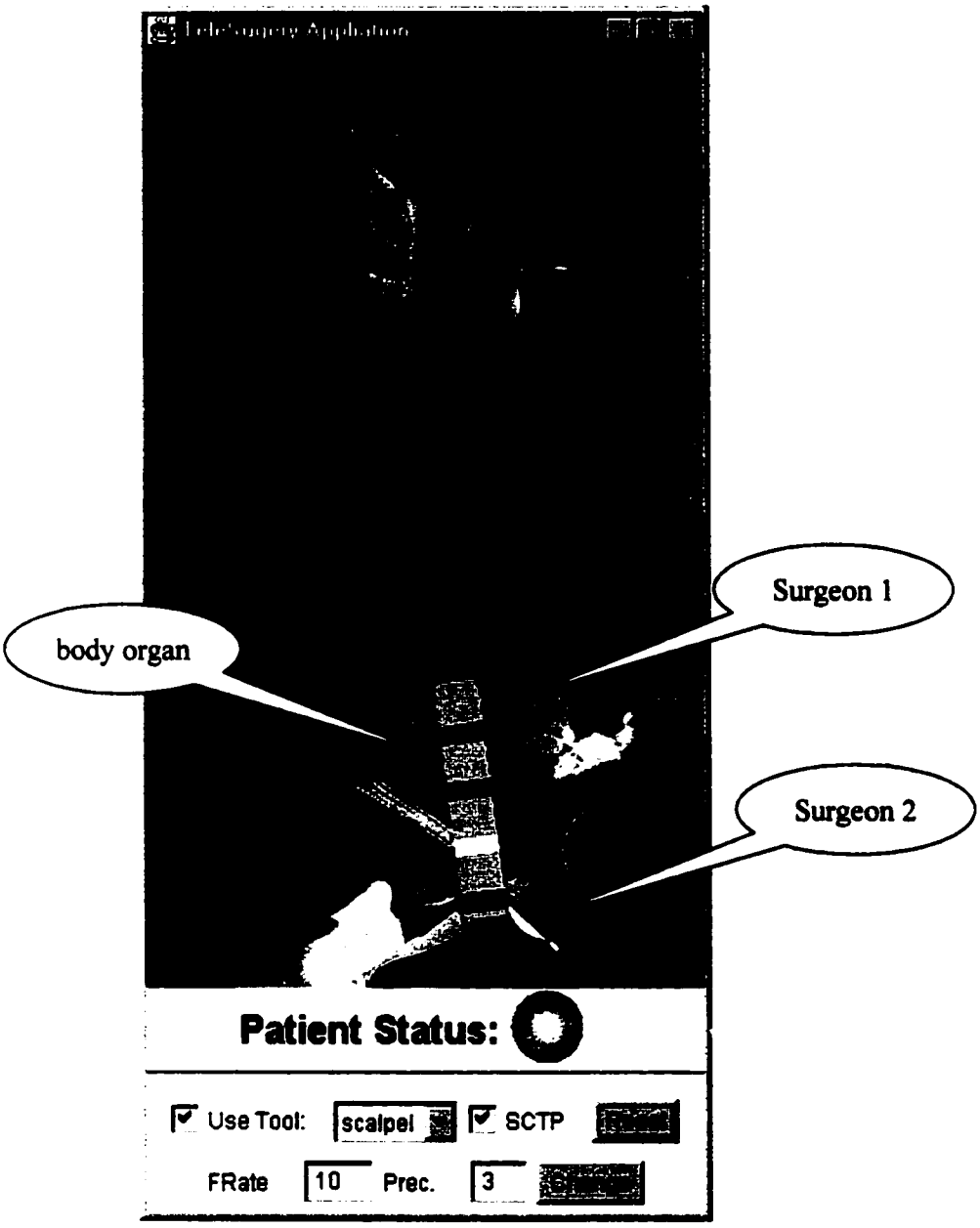


Figure 35. The TeleSurgery Application.

Although the operation itself is a fictional one, it certainly creates a closely-coupled environment to test the INVENTIST framework. The synchronous nature was confirmed by both the subjects and the results obtained.

To test the performance of INVENTIST under packet loss, it was necessary to create a mechanism to artificially drop packets from the network. This was achieved through a *Router* object, whose sole duty was to relay packets between any number of multicast networks, and drop packets randomly based on a given loss percentage. This was done in some of the classes of the ***inventist.test*** package shown in figure 34. Each surgeon was then put on a separate multicast address, with the router connecting the two multicast addresses (figure 36).

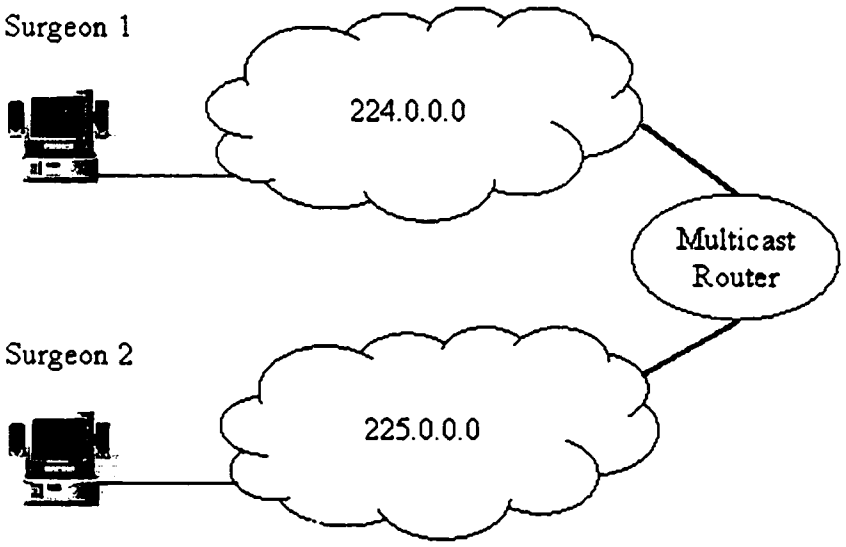


Figure 36. Multicast network setup.

The computers used in the test were Pentium II 333 MHz computers with 128 Meg RAM running Windows NT Workstation operating system.

6.5 Subjective Evaluation and Results

The above tele-surgery application was put to test between two users, one acting as surgeon 1 and the other acting as surgeon 2, whose duties were explained in 6.4. In the test, the occurrence of collaboration failures was monitored. This collaboration failure happens in three ways:

1- surgeon 2 cuts or stitches the wrong part of the organ because it does not correctly perceive the position of the organ;

2- surgeon 2 cuts/stitches the correct position of the organ, but surgeon 1 doesn't perceive that and still waits for surgeon 2 to do his duty;

3- surgeon 2 waits for surgeon 1 to move the organ to the next hot spot, surgeon 1 has already done so but surgeon 2 doesn't perceive that.

All of the above happen because of lost updates.

Each cutting act constitutes one collaboration "trial" which can fail or succeed, similarly each stitching act is a trial.

Before commencing the performance tests in presence of packet loss, the application was first tested with no packet loss; no failure was observed over 250 trials. This was a necessary test to make sure the failures that do occur are not due to the nature of the application, but are due to network loss.

The tests were performed for framerates of 5, 10, and 20; as well as loss rates of 10, 30, 50, and 70. Although the higher loss rates, specially the 70% loss, are very unlikely and unreal for a working network that wants to support collaborative applications, they were done for curiosity, just to see how SCTP behaves in this extremely lossy situation.

Each test was started with the non-SCTP protocol. Enough trials were performed to get either at least a few failures, or to reach 150 trials. 150 was chosen because it took about 6 minutes to do 150 trials, after which the participants needed to take a break since the tasks were quite focus-intensive. It should be noted that all interactions with the organ, such as moving, cutting, and stitching, were done as a mouse-drag operation, not a single-click operation. This was done in order to generate interaction streams that were larger than one update long.

The results of the tests are shown in the next two tables. Table 3 shows the number of failures out of the number of trials (failure/trial), while table 4 shows the amount of traffic generated during the tests.

Table 3. Collaboration failures. format: failure/trial.

framerate	loss %	SCTP	non-SCTP
5	10	0/150	6/150
	30	0/80	12/80
	50	0/50	18/50
	70	0/40	20/40
10	10	0/156	3/150
	30	0/100	7/100
	50	0/60	13/60
	70	0/50	20/50
20	10	0/150	1/150
	30	0/130	6/130
	50	0/60	9/60
	70	1/50	11/50

Table 4. Traffic. tx: transmitted; ACK: ack packets; reg: regular packets.

		SCTP					non- SCTP		
		A CK		r eg					
framerate	loss %	tx	lost	tx	lost	total (bits)	tx	lost	total (bits)
5	10	338	27	1105	121	**140348**	1061	107	**123076**
	30	230	66	638	186	**82288**	439	128	**50924**
	50	197	91	643	320	**81680**	337	171	**39092**
	70	228	154	1005	725	**124788**	287	196	**33292**
10	10	345	30	1567	154	**194192**	1538	147	**178408**
	30	292	86	1259	394	**156556**	1139	335	**132124**
	50	233	110	1053	532	**130536**	687	339	**79692**
	70	296	206	1559	1107	**191500**	665	476	**77140**
20	10	355	36	2468	232	**299068**	2343	249	**271788**
	30	388	115	2546	758	**309304**	2267	640	**262972**
	50	230	104	1467	716	**178452**	1134	586	**131544**
	70	300	216	2798	1945	**335368**	936	655	**108576**

6.5.1 Analysis

The lowest frame rate is 5 fps, because if an application goes below that frame rate, it already violates the 200 msec latency requirement. As the framerate of the application increases, the chance that specifically the last message is lost decreases due to the increased number of packets. This fact is reflected in table 3, where we can see that the number of failures decreases with increasing framerate.

Comparing the failure values between SCTP and non-SCTP, we can see that SCTP performs perfectly, under any framerate and any loss rate. This once again reinforces the design and architecture behind SCTP as it achieves its objective of supporting efficiently closely-coupled collaboration. On the other hand, the non-SCTP system produces

occasional failures which will disrupt the synchronous aspect of the collaboration session and will make it "uncomfortable" or "impossible" to collaborate from a human user's perspective. These failures occur even at low framerates and low packet loss rates.

In addition, as explained earlier, all actions of moving, cutting, and stitching were performed as mouse-drag operations, which translates into interaction streams with at least a few updates in each. But, if these actions were performed as mouse-clicks instead of mouse-drags, each stream would consists of only one update and the failure rate would have increased substantially, theoretically matching the packet loss rate.

Notice that one failure was observed for SCTP at 20 fps and 70% loss rate. This failure occurred when surgeon one moved the organ after surgeon 2 had cut the hot spot, but the last coordinate of the cutting was received by surgeon one when he was moving the organ already. This caused a cut to occur on a organ segment instead of hot spot. The reason for this late packet can be associated with the high number of packets in the network due to high framerate, and an unpractically high loss percentage. No network should have a 70% loss rate if it expects to carry out a synchronous collaboration session. Nonetheless, it seems that a "late update" phenomenon exists in SCTP at very high loss rate, which should not occur in the real world.

In terms of traffic, we can see from table 4 that SCTP produces more traffic, as expected. In a session with 2 users, which translates into 2 senders and 2 representatives, this traffic is higher than the non-SCTP traffic by an average of 10.7% at a loss rate of 10%, and by an average of 32% at a loss rate of 30%, over different frameworks. It is due to this extra

traffic that SCTP can provide the timely-reliable delivery of key updates, therefore efficiently supporting closely-coupled tasks in CVEs.

It should be noted that the number of updates generated per trial is not exactly the same between SCTP and non-SCTP tests. This is due to the human factor: participants' action of mouse drags are not the same for each drag operation. But over 150 trials, and using the same participants to do all tests will give us roughly the same number of updates produced per drag.

Chapter 7. Conclusions

In this thesis, several concepts with respect to synchronous collaboration in virtual environments were introduced. One was a new perspective that considers collaboration data to form an *Interaction Stream*, which consists of a burst of update messages with a final and critical update message. No other work has addressed the issue of collaboration at such level and in such a way.

It was shown by example that an interaction stream can typically be very short, and that a shared object often might not send update messages for an undetermined amount of time. In addition, the state of an object is not necessarily deterministic by predictive algorithms such as dead-reckoning. Therefore receiving the last state of an update at the end of a stream is crucial. Also, sometimes it is crucial to have reliability for some of the interim messages, in cases where the framerate of the application is too low, differential messages are used, or for congestion control purposes. Hence, reliability is not required for all messages, only a certain percentage of them. Other requirements were spelled out as low latency and minimal jitter. Low latency is important in general, but in closely coupled collaboration tasks latency becomes even more important in terms of the ability to efficiently collaborate.

All of the above factors and requirements gave rise to the design of SCTP, which was presented in detail. It was argued that using an ACK-based approach is essential,

specifically to efficiently support the action of tightly-coupled collaboration. The efficiency of the ACK-based approach was demonstrated theoretically, by simulation and by in practice.

The INVENTIST framework was then designed and implemented based on the proposed architecture. It was tested with actual subjects and proven to perform at the expected level. Although INVENTIST has currently been implemented as an extendible middleware, it can also run in standalone mode, fulfilling synchronous collaboration requirements for applications running directly on top of it. Further improvements to INVENTIST can be in the form of extending its design to address all CVE issues such as consistency, management, access control, and so on. But at this point, all the above issues are being addresses by some of the already-popular existing systems, and it is therefore more logical to provide synchronous collaboration services to these existing systems.

In my opinion, the reason that the proposed architecture works more efficiently than other approaches can be summarized in one sentence: "synchronous collaboration requires timely-reliability support for the correct perception of the last state of a shared object". This is a concept that other architectures either don't consider or don't support.

However, the ability to support synchronous collaboration was not achieved without a price. This price was found out to be traffic: SCTP generates more traffic compared to other approaches such as UDP multicast or SRTP. But this price is justifiable in applications where tightly-coupled collaboration is a necessity and cannot be compromised. Furthermore, the current trend in networking is the notion that "bandwidth is not a problem". Carrier and networking companies are starting to provide services that

allow quality of service demands with huge bandwidths on a per user basis. In fact, in the opening exhibition for the National Capital Institute of Telecommunications (NCIT) on May 21st 2000 in Ottawa, the MCRLab was asked to provide and demonstrate applications that consume huge bandwidths, in order to show that NCIT affiliated carrier companies are capable of supporting such bandwidths. But of course this higher bandwidth is not free, one has to pay for his/her demanded QoS parameters. In addition, there might be many groups of people running applications on the same network, leading to an accumulated high bandwidth. So technically speaking, huge bandwidth doesn't translate into "waste bandwidth", and it is still recommended to design bandwidth-friendly applications. But the impact of this trend is that in the near future, designers won't have to sacrifice application quality due to bandwidth. So SCTP's higher bandwidth consumption is not really a huge disadvantage.

In terms of applications, there are many applications that require synchronous collaboration services, such as telelearning, engineering design, manufacturing, tele-robotics, entertainment, and so on. The tele-surgery application used in this thesis was chosen because it had a good emphasis on "hard real-time" services that INVENTIST can provide. Although it might seem far-fetched, such tele-operations will be performed in the very near future, much nearer than one might think. There exist already "robotic" surgeons, which are controlled by a real surgeon. These robots provide what is referred to as *minimally invasive surgery* (MIS), which has the benefits of significantly reduced patient pain and trauma, shorter recovery times and convalescent periods, and overall improved outcome. One of the most famous of such robots is the *Zeus* surgery system created by the California-based Computer Motion Incorporated [67]. This robot allows

heart surgeries to be performed without opening the chest of a patient, otherwise known as open-heart surgery. It simply inserts its delicate robotic arms into the patient's body, closed-chest, and performs the heart surgery operation by receiving commands from a surgeon. The surgeon can view the operative site in either 3-D or 2-D, depending on his/her preference, as well as control the movements of the robot by either simple spoken commands or handles. Some pictures of Zeus in action are shown in figure 37.

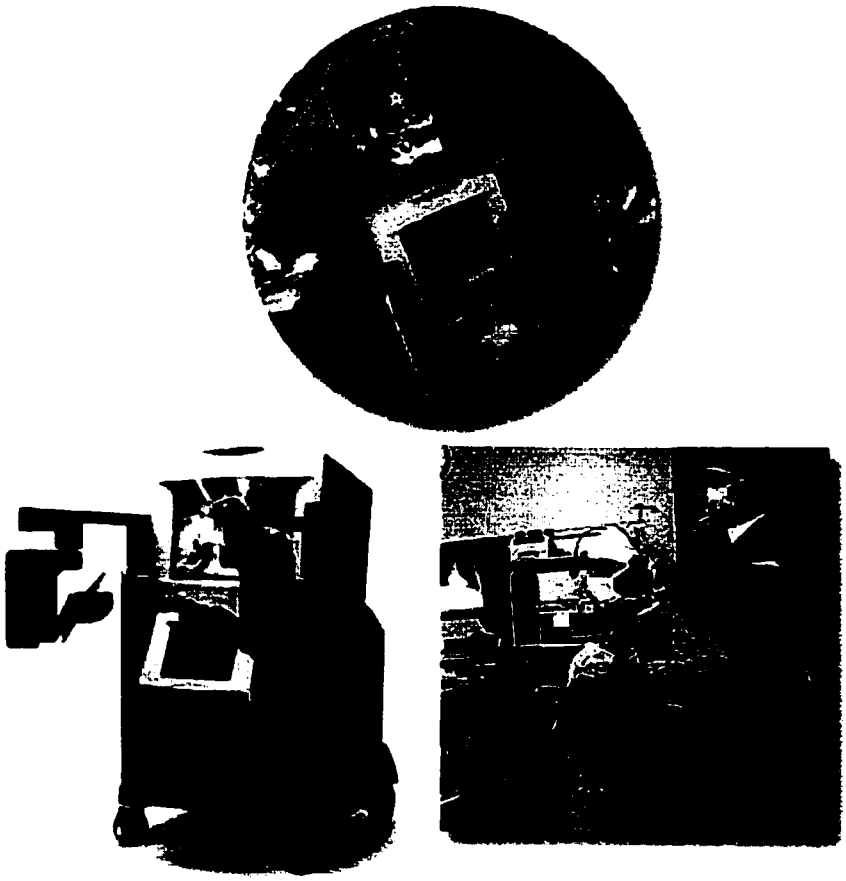


Figure 37. The ZEUS™ Surgery System. Photos courtesy of Computer Motion Inc.

Looking at apparatus such as this, one can foresee that very soon the operating surgeons can be in remote locations instead of being in the same room as the robot. If a network with extremely high degree of reliability and robustness is available, the surgeons can

send voice commands or handle movements from a remote location. The advantage of that is having specialized people operate on someone who lives in a location that does not otherwise have access to such specialized people, or in emergency situations.

The network in question has to be at least as reliable as the robot itself. In case of Zeus, the robot has self-diagnostic programs and it tests itself 400 times a second during an operation and alerts the operator if something is about to go wrong. Once a network with a high degree of robustness in terms of QoS fulfillment is built, then an architecture similar to INVENTIST can be used to perform even highly risky tasks.

VR and CVE systems of today are just beginning to become usable and practical. But there is still a long way to go until they actually affect our daily life, like computers and the Internet so easily do today. Still a lot of work has to be done to make VR systems easy to use, easy to navigate, robust, and graphically believable; or in other words "natural". Once that happens; i.e., virtual reality technology and its usage becomes as routine as computers and the Internet of today, it will create yet another revolution, at least as significant as the revolution created by computers and the Internet, if not more significant. The technology itself is very powerful, dangerously powerful in fact, in the sense that it can hide everything behind a natural user interface, an interface so natural that the user might not be able to distinguish it from reality. Like any other technology, this will have both advantages and disadvantages. On the positive side, this technology will create new ways of doing things that are not even imaginable right now. It will allow people to be virtually present in any location in the world at any time, without the need to physically move to that location: the ultimate goal of communications. On the negative side, it has the potential to create more isolation among real people, much more powerful

than that caused by the Internet, and health problems due to lack of physical movements in all aspects such as walking, exercising, travelling, and so on. But again, like any other technology, humans and the VR technology will adapt themselves to one another in order to minimize disadvantages and maximize the benefits. I for one am looking forward to the future when this technology is used widely.

References

[1] A. S. Tanenbaum, *Computer Networks*, Third Edition, Prentice Hall, New Jersey, 1996, pp. 536-539.

[2] A. Basu and S. J. Golestani, "Estimation of Receiver Round Trip Times in Multicast Communications", http://www.bell-labs.com/users/golestani/rtt.ps

[3] A. van Dam, "Immersive Virtual Reality for Scientific Visualization: A Progress Report", Keynote Address at the IEEE Virtual Reality Conference 2000, New Brunswick, NJ, U.S.A.

[4] B. Blau et al, "Networked Virtual Environments", Proc. ACM SGGRAPH, 1992, pp. 157-164.

[5] B. Fröhlish et al, "Physically-Based Manipulation on the Responsive Workbench", Proc. IEEE Virtual Reality Conference (IEEE VR 2000), New Brunswick, NJ, U.S.A., pp. 5-11.

[6] C. Diot and L. Gautier, "A Distributed Architecture for Multiplayer Interactive Applications on the Internet", IEEE Networks, Vol. 13, No. 4, July/August 1999, pp. 6-15.

[7] D. Rubenstein et al, "Real-Time Reliable Multicast Using Proactive Forward Error Correction", Proc. International Workshop on Network and Operating System Support for Digital Audio & Video (NOSSDAV '98), Cambridge, U.K., 1998.

[8] D. Dias et al, "Exploring JDSA, CORBA and HLA based MuTech's for Scalable Televirtal (TVR) Environments", Workshop on OO and VRML, VRML '98 conference, Monterey, California, February 1998.

[9] D. DeLucia and K. Obraczka, "A Multicast Congestion Control Mechanism for Reliable Multicast", IEEE Symposium on Computer and Communications, Athens, Greece, June/July 1998.

[10] D. Brutzman et al, "virtual reality transfer protocol (vrtp) Design Rationale", Proc. Workshops on Enabling Technology: Infrastructure for Collaborative Enterprises (WET ICE): Sharing A Distributed Virtual Reality, IEEE Computer Society, Cambridge Massachusetts, June 18-20 1997, pp. 179-186.

[11] Erramilli, and R Singh, "A Reliable and Effective Multicast Protocol for Broadband Broadcast Networks", Proc. ACM SIGCOMM, pp. 343-352, August 1987.

[12] G. Kessler, and L. Hodges, "A Network Communication Protocol for Distributed Virtual Environment Systems", Proc. IEEE Virtual Reality Annual International Symposium, 1996, pp. 214-221.

[13] G. Singh et al, "BrickNet: Sharing Object Behaviors on the Net", Proc. IEEE Virtual Reality Annual International Symposium, 1995, pp. 19-25.

[14] H. Ohzu and K. Habara, "Behind the Scenes of Virtual Reality: Vision and Motion", Proceedings of the IEEE, Vol. 84, No. 5, May 1996.

[15] Holbrook et al, "Log-based Receiver-Reliable Multicast for Distributed Interactive Simulation", Proc. ACM SIGCOMM '95, Cambridge, MA, August, 1995.

[16] IEEE Standard for Distributed Interactive Simulation, Application Protocols, IEEE 1278-1995.

[17] J. Barrus et al, "Locales and Beacons: Efficient and Precise Support for Large Multi-User Virtual Environments", Proc. IEEE Virtual Reality Annual International Symposium, 1996, pp. 204-213.

[18] J. Calvin and R. Weatherly, "An Introduction to the High Level Architecture (HLA) Runtime Infrastructure (RTI)," Proc. DIS Workshop on Standards for the Interoperability of Defense Simulations. Orlando, Florida, March 1996.

[19] J. Grudin, "Computer-Supported Cooperative Work: History and Focus", IEEE Computer, Vol. 27, No. 5, pp 19-26, May 1994.

[20] J. Leigh, "A Review of Tele-Immersive Applications in the CAVE Research Network", Proc IEEE International Conference on Virtual Reality (VR '99), Texas, March 1999.

[21] J. Leigh, A. Johnson and T. DeFanti, "CALVIN: an Immersimedia Design Environment Utilizing Heterogeneous Perspectives". Proc. IEEE International Conference on Multimedia Computing and Systems, IEEE Computer Society, Los Alamitos, Calif., 1996, pp. 20-23.

[22] J. Leigh et al, "Preliminary STAR TAP Tele-Immersion Experiments between Chicago and Singapore", High Performance Computing Asia Conference & Exhibition, Singapore, 1998.

[23] J. Oliveira et al, "Collaborative Virtual Environment Standards: A Performance Evaluation", Proc. IEEE Workshop on Distributed Interactive Simulations and Real-Time Applications (DIS-RT '99), Greenbelt, Maryland, October 1999.

[24] J. Signes, Y. Fisher, and A. Eleftheriadis, "BIFS Technical Description", Multimedia Systems, Standard, Networks, 1999 (to appear) http://flavor.ee.columbia.edu/mpeg4team/

[25] K. Hirota and T. Kaneko, "Real-Time Representation of Elastic Objects", Proc. IEEE Virtual Reality Conference (IEEE VR 2000), New Brunswick, NJ, U.S.A., May 2000, p. 285.

[26] K. Obraczka, "Multicast Transport Protocols: A Survey and Taxonomy", IEEE Communications, Vol. 36, No. 1, 1998, pp. 94-102.

[27] K. S. Park, *Effects of Network Characteristics and Information Sharing on Human Performance in COVE*, Master's thesis, Electronic Visualization Laboratory, University of Illinois at Chicago, 1997.

[28] K. S. Park and Robert V. Kenyon, "Effects of Network Characteristics on Human Performance in a Collaborative Virtual Environment", IEEE Virtual Reality, Houston, Texas, March 1999.

[29] L. Rizzo, "Effective Erasure Codes for Reliable Computer Communications", ACM Computer Communications Review, Vol. 27, No 2, 1997, pp. 24-36.

[30] M.M. Wloka, "Lag in Multiprocessor VR", Presence: Teleoperators and Virtual Environments (MIT Press), Vol. 4, No. 1, Spring 1995.

[31] M Pullen, "Reliable Multicast Network Transport for Distributed Virtual Simulation", Proc. IEEE Workshop on Distributed Interactive Simulations and Ral-Time Applications (DIS-RT '99), Greenbelt, Maryland, October 1999.

[32] M. Pullen et al, "Limitations of Internet Protocol Suite for Distributed Simulation in the Large Multicast Environment", RFC 2502, February 1999.

[33] M. Pullen, and V. Laviano, "A Selectively Reliable Transport Protocol for Distributed Interactive Simulation", Proc. 13th Workshop on Standards for Distributed Interactive Simulation, September 1995.

[34] M.R. Macedonia et al, "Exploiting Reality with Multicast Groups: A Network Architecture for Large-Scaled Virtual Environments", Proc. IEEE Virtual Reality Annual International Symposium, 1995, pp. 2-10.

[35] M.R. Macedonia, and M.J. Zyda, "A Taxonomy for Networked Virtual Environments", IEEE Multimedia Magazine, January-March 1997, pp. 48-56.

[36] M. Yajnik et al, "Packet Loss Correlation in the MBone Multicast Network", Proc IEEE Global Internet Conference, November 1996, London, U.K.

[37] M. Yamato et al, "A Delay Analysis of Sender-Initiated and Receiver Initiated Reliable Multicast Protocols", Proc. IEEE Conference on Computer Communications (INFOCOM' 97), April 1997.

[38] Multimedia Communication Forum Inc., "Multimedia Communication Quality of Service", MMCF document MMCF/95-010, Approved Rev 1.0, September 24, 1995.

[39] N.D. Georganas, "Advanced Distributed Simulation and Collaborative Virtual Environments", CRC Internal Report, Dec.1997.

[40] N.D. Georganas et al, "Distributed Virtual Environments for Training and Telecollaboration", Proc. IEEE Instrumentation and Measurement Technology Conference (IMTC '99) Venice, Italy, May 24-26, 1999.

[41] Olof Hagsand, "Interactive Multi-user VEs in the DIVE System", IEEE Multimedia, pp. 30-39, Spring 1996

[42] R.C. Waters and J. Barrus, "The Rise of Shared Virtual Environments", IEEE Spectrum, March 97.

[43] R.C. Waters, D. Anderson, and S. Schwenke, "The Interactive Sharing Transfer Protocol Version 1.0", Mitsubishi Electric Research Laboratory Technical Report, MERL-TR-97-10, October 1997.

[44] R.C. Waters et al, ``Design of the Interactive Sharing Transfer Protocol", Proc. WET ICE '97 --IEEE Sixth Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises, IEEE Computer Society Press, Los Alamitos CA, 1997.

[45] R. G. Kermode, "Scoped Hybrid Automatic Repeat reQuest with Forward Error Correction", Proc ACM SIGCOMM '98, Vancouver Canada, 1998, pp. 278-289.

[46] S. Floyd et al, "A Reliable Multicast Framework for Light-Weight Sessions and Application Level Framing", IEEE/ACM Transactions on Networking, November 1996.

[47] S. J. Golestani and K. Sabnani, "Fundamental Observations on Multicast Congestion Control in the Internet", Proc. Infocom '99, 1999.

[48] S. Gray, "Virtual Reality in Virtual Fashion", IEEE Spectrum, Vol. 35, No. 2, February 1998, pp. 18-25.

[49] S. Moezzi, "Immersive Telepresence", IEEE Multimedia, Vol. 4, No. 1, Winter 1997, P. 17.

[50] S. Pingali, D. Towsley, and J. Kurose, " A Comparison of Sender-Initiated and Receiver-Initiated Reliable Multicast Protocols," Proc. ACM SIGMETRICS, Hashvilee, TN, May 1994.

[51] S. Seidensticker et al, "Scenarios and Appropriate Protocols for Distributed Interactive Simulation", Internet draft, draft-myjak-lsma-scenarios-02, march 1997.

[52] S. Shirmohammadi et al, "Applet-Based Telecollaboration: A Network-Centric Approach", IEEE Multimedia Magazine, Volume 5, Number 2, April-June 1998, pp. 64-73.

[53] S. Singhal, and M. Zyda, Networked Virtual Environments, ACM Press, NY, NY, 1999, p. 143.

[54] T. Funkhouser, "RING: A Client-Server System for Multi-User Virtual Environments.", Proc. ACM SIGGRAPH, 1995, pp. 85-92.

[55] T. Funkhouser, "Network Topologies for Scalable Multi-User Virtual Environments", Proc. IEEE Virtual Reality Annual International Symposium, 1996, pp. 222-228.

[56] V. Laviano, and M. Pullen, "Selectively Reliable Transmission Protocol", Internet Draft draft-laviano-srtp-01, August 1997.

[57] W. Bricken et al, "The VEOS Project", Technical Report, Human Interface Technology Laboratory, University of Washington, 1993.

[58] CAVERN , http://www.evl.uic.edu/spiff/covr

[59] COVEN, http://chinon.thomson-csf.fr/projects/coven

[60] DIVE, http://www.sics.se/dive/dive.html

[61] High Level Architecture (HLA), http://www.dmso.mil/hla

[62] Living Worlds Proposal, http://www.vrml.org/WorkingGroups/living-worlds

[63] OnLive! Traveler, http://www.onlive.com/prod/trav

[64] Open Community, http://www.meitca.com/opencom

[65] Telelearning Network of Centers of Excellence, http://www.telelearn.ca

[66] VRML Consortium, http://www.vrml.org

[67] ZEUS Robotic Surgical System, Computer Motion Incorporated, http://www.computermotion.com/zeus.html

[68] OPNET simulation tool, MIL3 Inc., http://www.mil3.com.

APPENDIX: The INVENTIST Application Programming Interface (API)

The following packages have been developed.

Packages	
inventist	
inventist.com	
inventist.com.sctp	
inventist.surgery	
inventist.test	
inventist.util	

Notice that the API consists only of packages *inventist*, *inventist.com*, and *inventist.util*. *inventist.com.sctp* is my implementation of the communications module based on SCTP and requires no API since it will not be accessed by the applications. *inventist.surgery* contains the classes for the tele-surgery application and *inventist.test* contains classes to test INVENTIST, as well as the multicast router, none of which are required to be accessed by a higher level application.

The following pages show the API for INVENTIST based on packages.

inventist

Class Inventist

```
java.lang.Object
  |
  +--inventist.Inventist
```

Direct Known Subclasses:
InventistImpl, InventistImpl, surgery

---

public abstract class **Inventist**
extends java.lang.Object

INVENTIST: INception of Virtual ENvironments' TIghtly-Synchronous Tasks. This is the main object of the system. It takes care of the Interaction Stream concept. It uses the Steram Handler Object to take care of timers for shared objects, and it uses the Comm object for network communications. Any system wanting to use INVENTIST must extend this abstract class. See my Ph.D. Thesis for conceptual details. (c) Shervin Shirmohammadi, Feb. 11 2000, MCRLab, University of Ottawa

---

Field Summary

protected Comm	**commModule**
int	**MaxIndex**
protected SharedObject[]	**sharedObjects**
protected StreamHandler	**stHandler**

Constructor Summary

Inventist()

Method Summary

void	**addSharedObject**(int index) Creates a new Shared Object with the given index.
void	**deleteSharedObject**(int index) Deletes the Shared Object indicated by index.
SharedObject	**getSharedObject**(int index) Returns the Shared Object indicated by index.
void	**init**()
void	**joinMulticastGroup**(java.lang.String address, int port) Joins a given multicast group.
void	**leaveMulticastGroup**(java.lang.String address) Leaves a given multicast group.
void	**sendKeyUpdate**(UpdateMessage um, int index) Forces an update message to be sent as a KEY for the Shared object indicated by the index, without regard for the key update frequency.
void	**sendUpdate**(UpdateMessage um, int index) Sends an update message for the Shared object indicated by the index.
void	**setAutomaticFrequency**(long t) Automatates the process of calculating which updates are key for shared object indicated by index.
void	**setCommModule**(Comm commMod) Sets the communication module.
void	**setKeyFrequency**(int n) Sets the frequency of sending key updates for all shared objects.
void	**setKeyFrequency**(int n, int index) Sets the frequency of sending key updates for shared object indicated by index.
void	**setKeyFrequency**(long t) Sets the frequency of sending key updates for all shared objects.
void	**setKeyFrequency**(long t, int index) Sets the frequency of sending key updates for shared object indicated by index.
abstract void	**updateReceived**(UpdateMessage um, int index) Callback: is called when an update message for the Shared Object indicated by index is received from the network.
void	**Wait**(long t)

Methods inherited from class java.lang.Object

, clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

MaxIndex

```
public final int MaxIndex
```

---

sharedObjects

```
protected SharedObject[] sharedObjects
```

---

commModule

```
protected Comm commModule
```

---

stHandler

```
protected StreamHandler stHandler
```

Constructor Detail

Inventist

```
public Inventist()
```

Method Detail

init

```
public void init()
```

---

setCommModule

```
public void setCommModule(Comm commMod)
```

Sets the communication module.

---

sendUpdate

```
public void sendUpdate(UpdateMessage um,
                       int index)
```

Sends an update message for the Shared object indicated by the index.

sendKeyUpdate

```
public void sendKeyUpdate(UpdateMessage um,
                          int index)
```

Forces an update message to be sent as a KEY for the Shared object indicated by the index, without regard for the key update frequency.

---

updateReceived

```
public abstract void updateReceived(UpdateMessage um,
                                    int index)
```

Callback: is called when an update message for the Shared Object indicated by index is received from the network.

---

joinMulticastGroup

```
public void joinMulticastGroup(java.lang.String address,
                               int port)
```

Joins a given multicast group.

---

leaveMulticastGroup

```
public void leaveMulticastGroup(java.lang.String address)
```

Leaves a given multicast group.

---

getSharedObject

```
public SharedObject getSharedObject(int index)
```

Returns the Shared Object indicated by index. Returns null if An object with the given index doesn't exist.

---

setKeyFrequency

```
public void setKeyFrequency(int n)
```

Sets the frequency of sending key updates for all shared objects. Every nth update is sent as key.

---

setKeyFrequency

```
public void setKeyFrequency(long t)
```

Sets the frequency of sending key updates for all shared objects. Every t milliseconds an update is sent as key.

---

setKeyFrequency

```
public void setKeyFrequency(int n,
                            int index)
```

Sets the frequency of sending key updates for shared object indicated by index. Every nth update is sent as key.

---

setKeyFrequency

```
public void setKeyFrequency(long t,
                            int index)
```

Sets the frequency of sending key updates for shared object indicated by index. Every t milliseconds an update is sent as key.

---

setAutomaticFrequency

```
public void setAutomaticFrequency(long t)
```

Automatates the process of calculating which updates are key for shared object indicated by index. Also sets the frequency for the automatic key generation process.

---

addSharedObject

```
public void addSharedObject(int index)
```

Creates a new Shared Object with the given index.

---

deleteSharedObject

```
public void deleteSharedObject(int index)
```

Deletes the Shared Object indicated by index.

Wait

```
public void Wait(long t)
```

---

inventist

Class SharedObject

```
java.lang.Object
  |
  +--inventist.SharedObject
```

public class **SharedObject**
extends java.lang.Object

Constructor Summary

SharedObject(int index)

Method Summary

int	**getLastKeySequenceNum**() Returns the last key sequence number for this object
int	**getStreamNum**() Returns the current stream number for this object
void	**setLastKeySequenceNum**(int seq) Sets the last key sequence number for this object
void	**setStreamNum**(int strm) Sets the current stream number for this object

Methods inherited from class java.lang.Object

, clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

SharedObject

public **SharedObject**(int index)

Method Detail

getLastKeySequenceNum

```
public int getLastKeySequenceNum()
```

Returns the last key sequence number for this object

---

setLastKeySequenceNum

```
public void setLastKeySequenceNum(int seq)
```

Sets the last key sequence number for this object

---

getStreamNum

```
public int getStreamNum()
```

Returns the current stream number for this object

---

setStreamNum

```
public void setStreamNum(int strm)
```

Sets the current stream number for this object

---

inventist

Class StreamHandler

```
java.lang.Object
  |
  +--inventist.StreamHandler
```

public class **StreamHandler**
extends java.lang.Object
implements TimerListener

Constructor Summary

StreamHandler(Inventist inv, long timeout)

Method Summary

void	**newUpdate**(UpdateMessage um, int index) Indicates a new update for object index has arrived.
void	**setTimeout**(long t) Sets the time-out value
void	**stop**() Stop stream handler for all objects.
void	**stop**(int index) Stop stream handler for object index.
void	**timeExpired**(UpdateMessage um, int index, int seq) Callback from the timer.

Methods inherited from class java.lang.Object

, clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

StreamHandler

```
public StreamHandler(Inventist inv,
                     long timeout)
```

Method Detail

setTimeout

```
public void setTimeout(long t)
```

Sets the time-out value

---

newUpdate

```
public void newUpdate(UpdateMessage um,
                      int index)
```

Indicates a new update for object index has arrived.

---

timeExpired

```
public void timeExpired(UpdateMessage um,
                        int index,
                        int seq)
```

Callback from the timer.

Specified by:

timeExpired in interface TimerListener

---

stop

```
public void stop()
```

Stop stream handler for all objects. No more automatic key estimation will be done.

---

stop

```
public void stop(int index)
```

Stop stream handler for object index. No more automatic key estimation will be done.

---

inventist

Class UpdateMessage

```
java.lang.Object
  |
  +--inventist.UpdateMessage
```

---

public class **UpdateMessage**
extends java.lang.Object

Constructor Summary

UpdateMessage()
UpdateMessage(int x, int y, int z, double rot)

Method Summary

double	**getRotation**()
int	**getX**()
int	**getY**()
int	**getZ**()
void	**setRotation**(double rot) Set the orientation.
void	**setXYZ**(int x, int y, int z) Set the cartesian coordinates for the update message.

Methods inherited from class java.lang.Object

, clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

UpdateMessage

```
public UpdateMessage()
```

UpdateMessage

```
public UpdateMessage(int x,
                     int y,
                     int z,
                     double rot)
```

Method Detail

setXYZ

```
public void setXYZ(int x,
                   int y,
                   int z)
```

Set the cartesian coordinates for the update message.

---

setRotation

```
public void setRotation(double rot)
```

Set the orientation.

---

getX

```
public int getX()
```

---

getY

```
public int getY()
```

---

getZ

```
public int getZ()
```

getRotation

```
public double getRotation()
```

---

inventist.com

Class Comm

```
java.lang.Object
   |
   +--inventist.com.Comm
```

Direct Known Subclasses:
Comm, CommImpl

---

public abstract class **Comm**
extends java.lang.Object

Field Summary

protected Inventist	**inv**

Constructor Summary

Comm()

Method Summary

void	**incomingUpdate**(UpdateMessage um, int index) Callback to inventist.
void	**init**(Inventist inv) Initialize the communications module.
abstract void	**join**(java.lang.String address, int port) Join a session.
abstract void	**leave**(java.lang.String address) Leave a session.
abstract void	**sendUpdate**(UpdateMessage um, boolean key, int index) send an update for object index.
abstract void	**setTimeOut**(long timeout) Set re-transmission timeout.

Methods inherited from class java.lang.Object
, clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

inv

```
protected Inventist inv
```

Constructor Detail

Comm

```
public Comm()
```

Method Detail

init

```
public void init(Inventist inv)
```

Initialize the communications module.

---

sendUpdate

```
public abstract void sendUpdate(UpdateMessage um,
                                boolean key,
                                int index)
```

send an update for object index. key indicates whether the update is a key or not.

---

incomingUpdate

```
public void incomingUpdate(UpdateMessage um,
                           int index)
```

Callback to inventist.

---

join

```
public abstract void join(java.lang.String address,
                          int port)
```

Join a session.

---

leave

```
public abstract void leave(java.lang.String address)
```

Leave a session.

---

setTimeOut

```
public abstract void setTimeOut(long timeout)
```

Set re-transmission timeout.

---

inventist.util

Interface TimerListener

All Known Implementing Classes:

CommImpl, StreamHandler, surgery

public abstract interface **TimerListener**

This interface must be implemented by objects that wish to be notified by a timer when time expires.

Method Summary

void	**timeExpired**(UpdateMessage um, int index, int seq)

Method Detail

timeExpired

```
public void timeExpired(UpdateMessage um,
                        int index,
                        int seq)
```

This method is called by the timer when time has expired.

inventist.util

Class Timer

```
java.lang.Object
  |
  +--java.lang.Thread
        |
        +--inventist.util.Timer
```

public class **Timer**
extends java.lang.Thread
This class is a simple timer.

Fields inherited from class java.lang.Thread
MAX_PRIORITY, MIN_PRIORITY, NORM_PRIORITY, values

Constructor Summary

Timer(long t, UpdateMessage um, int index, int seq)

Method Summary

void	**addTimerListener**(TimerListener tl)
void	**cancelTimer**()
void	**run**()

Methods inherited from class java.lang.Thread
, activeCount, checkAccess, countStackFrames, currentThread, destroy, dumpStack, enumerate, getContextClassLoader, getName, getPriority, getThreadGroup, interrupt, interrupted, isAlive, isDaemon, isInterrupted, join, join, join, resume, setContextClassLoader, setDaemon, setName, setPriority, sleep, sleep, start, stop, stop, suspend, toString, yield

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

Constructor Detail

Timer

```
public Timer(long t,
             UpdateMessage um,
             int index,
             int seq)
```

t: amont of timeout in msec.

um: update message for which timeout is being taken.

index: object for which timeout is being taken.

seq: sequence number for which timeout is being taken.

Method Detail

addTimerListener

```
public void addTimerListener(TimerListener tl)
```

Adds the given timerlistener object to this timer's list of wake-up objects.

---

run

```
public void run()
```

Overrides:

run in class java.lang.Thread

---

cancelTimer

```
public void cancelTimer()
```
