CAVERN AND A UNIFIED APPROACH TO SUPPORT REALTIME NETWORKING AND PERSISTENCE IN TELEIMMERSION

BY

JASON LEIGH
B.S., Computer Science, University of Utah, 1988
M.S., Computer Science, Wayne State University, 1991

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical Engineering
and Computer Science
in the Graduate College of the
University of Illinois at Chicago, 1998

Chicago, Illinois

To my parents Francis and Betty Leigh,

and my grandfather Lee Tak Hung.

ACKNOWLEDGMENTS

I owe a huge debt of gratitude to my committee, who have also been my mentors over the years: Tom DeFanti, Dan Sandin, Tom Moher, Bob Grossman, Andy Johnson, and Larry Smarr.

I would also to like to thank my closest friends Karl Fitzpatrick, Andy Johnson (a second time) and Christina Vasilakis for being great companions through life.

Finally I would like to thank the staff members of EVL, in particular Maggie Rawlings, Dana Plepys, and Maxine Brown who have supported me since I first set foot in EVL, N-years ago.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

CHAPTER **PAGE**

LIST OF FIGURES

LIST OF FIGURES (Continued)

LIST OF FIGURES (Continued)

SUMMARY

Teleimmersion[1] (also referred to as Collaborative Virtual Reality) is the unification of teleconferencing, and collaborative immersion in virtual environments in order to provide the "truest" sense of co-presence. That is, the sense that one is present in the same physical space with one's collaborators. Collaborators are not only able to see and talk to each other face-to-face but are able to naturally convey gesture and body language.

Teleimmersion is currently one of the most challenging areas of research in Virtual Reality (VR.) Networking adds a new dimension to many areas of VR research. For example human-factors research in VR traditionally focuses on the development of more natural means of manipulating virtual objects and traversing virtual landscapes. However *collaborative* manipulation forces the consideration of how participants should interact with each other in a shared space, in addition to how co-manipulated objects should behave. There are also questions of how participants should be represented in the collaborative environment; how to effectively transmit non-verbal cues that real-world collaborators so casually use; how to best transmit video and audio via a channel that allows both public addressing as well as private conversations to occur; and how to sustain a virtual environment even when all its participants have left.

Naturally Teleimmersion poses new challenges to traditional areas of networking and databases as well. Teleimmersive environments (TIEs) require an unconventionally broad range of net-

[1]The term Teleimmersion was coined by Thomas A. DeFanti and Daniel J. Sandin of the Electronic Visualization Laboratory at the University of Illinois at Chicago

working, database and graphics capabilities to realize and sustain. This vast range makes the rapid construction of rich TIEs difficult. Past attempts at building networking and database architectures for Teleimmersion have resulted in ad-hoc solutions that are specifically designed to solve a small range of problems and hence little reusability was possible. Nevertheless from these early attempts patterns began to emerge. In particular: the realization that the application domain can significantly impact the kind of networking topology and protocols needed to distribute the data; the realization that a tighter integration between networking and databases is needed to support long term teleimmersive applications; and the realization that multiple separable software layers are needed to allow application developers to rapidly create new teleimmersive applications, as well as to integrate teleimmersive capabilities into existing non-teleimmersive applications.

This dissertation's main contribution is in a) proposing CAVERNsoft, a broad conceptual solution to the problem; and b) proposing and implementing a software foundation for CAVERNsoft called the Information Resource Broker (IRB.) The IRB explores the feasibility and effectiveness of using a persistent, distributed shared memory, to support data distribution in Teleimmersion.

In the chapters to follow I will begin by more deeply examing the problems of supporting Teleimmersion by first illustrating teleimmersive concepts through a number of examples. These examples will form the basis for determining the set of characteristics that impact teleimmersive environments. Following this a proposed solution will be presented along with a detailed description of its implementation and its evaluation.

CHAPTER 1

INTRODUCTION TO TELEIMMERSION

The following scenarios describe representative Teleimmersive Environments (TIEs) in several domains. Although these scenarios may not fully represent all possible scenarios that will arise in the future of Teleimmersion, they are chosen because they are either historically illustrative- or are currently illustrative- of problems that are actively being researched.

One constraint this dissertation attempts to impose on the scenarios, is that they involve tasks that would benefit from a solution in Teleimmersion over simply non-collaborative VR or 3D workstation computer graphics. For example, simple audio/video teleconferencing alone is not considered a scenario that can significantly benefit from the use of Teleimmersion. However collaborative work that depends on the spatial qualities of VR (such as collaborative architectural design) in addition to teleconferencing, as part of its solution, *is* considered a good candidate for a Teleimmersive solution.

1.1 Collaborative Design and Engineering

1.1.1 Caterpillar Collaborative Design

Collaborative design work in VR typically involves a small group of users, either synchronously or asynchronously, engaged in the construction, and manipulation of objects in the virtual world. Since the interfaces for three-dimensional modeling in VR are still relatively imprecise compared to 2.5D CAD packages, most of the collaborative tasks in collaborative design involve evaluations of the design, and to a lesser degree, redesign or brainstorming for new design possibilities(1; 2; 3).

The National Center Supercomputing Applications (NCSA) has been working with Caterpillar Belgium S.A., to develop a system to allow remotely located engineers to work together on vehicle design review and redesign(4). Remote collaboration is necessary here because the eventual system will be used by Caterpillar engineers in the U.S. and Europe who must jointly design Caterpillar vehicles so that they meet customer demands and safety requirements for both markets. For example, European safety standards require a roading fender to be added to the basic vehicle design. The collaborative VR system allows engineers to evaluate rearward visibility from a viewpoint in the virtual cab of the vehicle. Virtual co-presence allows one designer to manipulate the fender while another designer watches for its effect on visibility.

To support user-to-user communication, publicly available audio and video teleconferencing tools (*vat* and *nv* respectively) were modified to work with the CAVE virtual environment(5). Video images from each participant were texture-mapped onto the surface of a rectangular box to establish their presence in the environment. The 3D models of the Caterpillar vehicles that are used in the collaboration are first duplicated at every site. Then an unreliable multicast data stream is used to distribute information about the participants and changes in the models to all the other participants.

1.1.2 CALVIN - Collaborative Architectural Layout Via Immersive Navigation

CALVIN (2; 1; 3) is a TIE that allows multiple users to synchronously and asynchronously experiment with architectural room layout designs in the CAVE (Figure 1.)

Participants are able to move, rotate, and scale architectural design pieces such as walls and furniture. Participants may work as either "mortals" who see the world life-sized (classically known as an "inside-out" view), or as "deities" ("outside-in" view) who see the world as if it were a miniature model. Deities by virtue of their enlarged size relative to the environment, tend to tower above the scene and are better at performing gross manipulations on objects. Mortals on the otherhand are at the same scale as the environment, and are hence better able to perform fine manipulations.

Asynchronous access allows designers to enter the space whenever inspiration strikes them, rather than requiring them to wait to schedule formal meetings, which can be particularly difficult if the participants are located at opposite parts of the world with significant timezone differences. In fact CALVIN already provides interfaces for bilingual (Japanese and English) interaction.

Participants are able to save versions of the design as the collaboration progresses. When participants re-enter the environment at a later time, the most recently saved version is automatically loaded. If on the otherhand the participant re-loads a different version of the design, CALVIN will record successive designs as a new branch in the version tree.

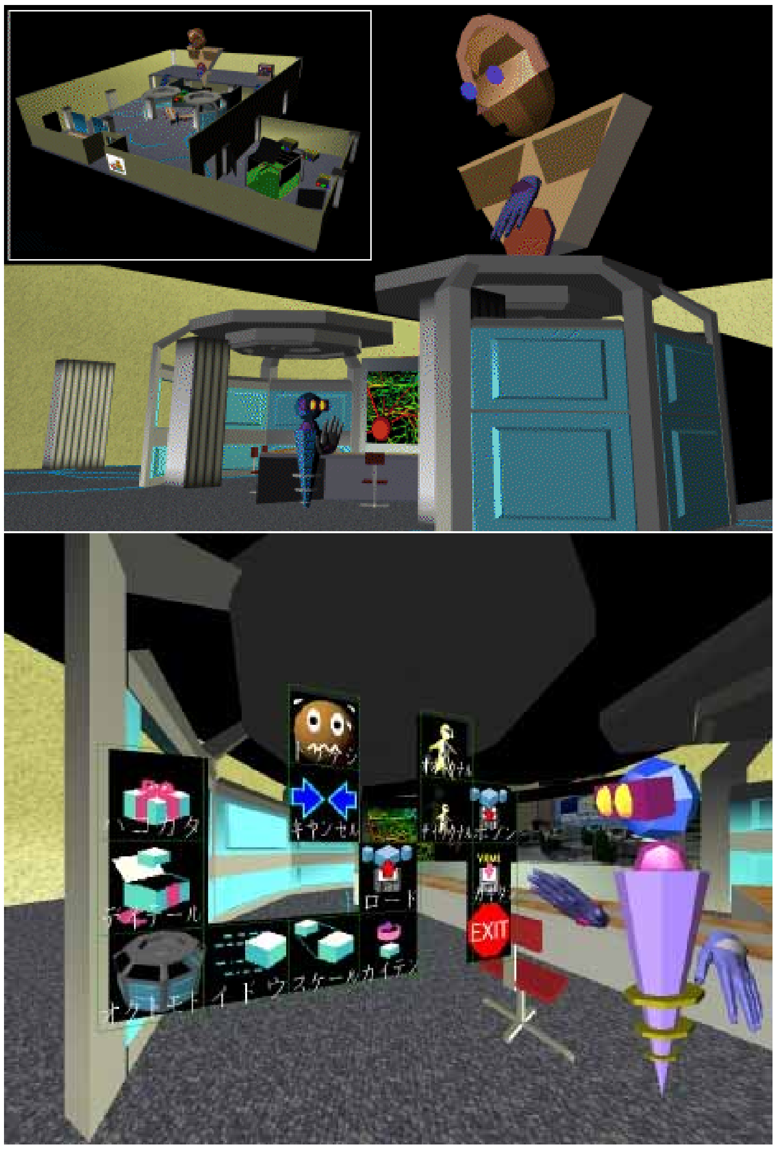


Figure 1. CALVIN: a collaborative design environment for architectural layout. The scene shows two avatars (a tall one and a short one) viewing the space at different perspectives. The top lefthand inset of the top image is a zoomed-out view of the entire design space. The lower image shows one of the avatars using CALVIN's Japanese interface.

CALVIN employees a shared variable model of a distributed shared memory (DSM) system to eliminate the need of the programmers to develop specific protocols for network communication. The DSM itself uses a reliable protocol and a centralized sequencer to guarantee consistency in all clients. C++ classes representing networked versions of floats, integers and character arrays are provided so that assignment to variable instantiations of these classes automatically shares the information with all the remote clients.

These networked variables are used to send data such as the state of objects in the world and user-tracker information. Tracker information is sent so that avatars can be drawn in the place of participants in the virtual scenes. Position as well as orientation data from the user's hand and head are transmitted so that fundamental gestures such as nodding, pointing, and waving can be communicated through the avatars.

Although the task of world synchronization is greatly simplified by the centralized sequencer, the transmission of tracker information over such a reliable channel can introduce latencies-especially when synchronizing between the participant's real location and their avatar's location. This is acceptable for small relatively closely located working groups where the network traffic and latency is relatively low but is unsuitable for larger and more distant groups of participants dispersed over the internet. In fact, to transmit audio/video signals between sites, the shared memory system is bypassed with point-to-point raw ATM streams which are able to support teleconferencing at NTSC resolution and at 30 frames per second.

Finally, in CALVIN when two or more participants simultaneously modify an object, a "tug-of-war" occurs where the object appears to jump back and forth between two positions, eventually remaining at the position given to it by the last person holding onto it. This problem can be alleviated by using a locking scheme, but this was intentionally **not** done. In VR, where emphasis is placed on natural interaction, it would be unnatural if the user had to lock an object before picking it. The presence of avatars in combination with audio communication (the most important of the communication channels to provide) compensated for the lack of strict floor control and database locking. For example, the declaration: "I'm going to move this chair" combined with the visual cue of an avatar standing next to a chair and pointing at it, alerts other users that this user is about to grab that chair.

1.2 Collaborative Training

1.2.1 Military Simulations

The earliest teleimmersive systems were military-based applications such as SIMNET and NPSNET (6; 7; 8). SIMNET is a standard for distributed interactive simulations developed by DARPA beginning in 1985. The purpose of SIMNET was to facilitate early phases of training at a cost far below the expense of conducting real battlefield exercises. A SIMNET participant may be wearing a head-mounted display and standing on a tread-mill to train as a foot-soldier. Alternatively another SIMNET participant may be sitting in a tank simulator. Typically, SIMNET expects hundreds of participants to be engaged in the simulation at the same time. To reduce the bandwidth and the effects of latency needed to sustain this degree of scalability

SIMNET uses a technique called dead-reckoning to predict the location of participants at any instant in time based on their previous reported position, velocity and acceleration.

As SIMNET was designed primarily for military simulation, its underlying unit of data transmission (called a Protocol Data Unit- PDU for short) specifically contains encodings for military entities (such as tanks and airplanes.) DIS (Distributed Interactive Simulation) is a newer and more ambitious simulation standard (IEEE 1278) that is based on SIMNET but allows for greater complexity and realism. For example: SIMNET uses a flat terrain whereas DIS accounts for the curvature of the Earth. SIMNET is oriented towards terrain and the sky above it whereas DIS encompasses all areas of potential military activity including below the ocean and in space.

1.2.2 NICE - Narrative Immersive Constructionist/Collaborative Environments

The NICE group is building a collaborative environment in the form of a virtual island for young children (approximately 6-8 years of age)(9; 10). In the center of this island the children can tend a virtual garden. The children, represented by avatars, collaboratively plant, grow, and pick vegetables and flowers. They ensure that the plants have sufficient water, sunlight, and space to grow, and need to keep a look out for hungry animals which may sneak in and eat the plants. The children can shrink down to the size of a mouse and crawl under the garden to see the root system, and can talk with the other remotely located children or other characters in the scene. The children are able to modify the parameters of this small ecosystem to see how it affects the health of the garden (Figure 2.)

NICE's architecture is based on the techniques derived from CALVIN in that a central server is used to maintain consistency across all the participating virtual environments. Whereas CALVIN solely used a reliable connection to synchronize state information, NICE used an unreliable protocol (either multicasting or UDP) to share avatar information from magnetic trackers; and a reliable socket connection to share world state information and to dynamically download models from WWW servers using the HTTP 1.0 protocol.

Both multicasting and UDP were provided to deliver tracker data, as it was not always possible to acquire the administrative privileges to conveniently erect multicast tunnels between distant remote sites. Hence a number of interconnected NICE "smart-repeaters" were deployed at various remote sites that allowed the use of multicasting amongst clients at localized sites but UDP for repeating packets between remote locations. In addition, to prevent faster clients from overwhelming slower clients with data, the smart-repeaters performed dynamic filtering of data based on the throughput capabilities of the clients. Using this scheme participants running on high speed networks, have been able to collaborate with participants running on slower 33Kbps modem lines.

NICE's virtual environment is persistent. That is, even when all the participants have left the environment and the virtual display devices have been switched off, the environment

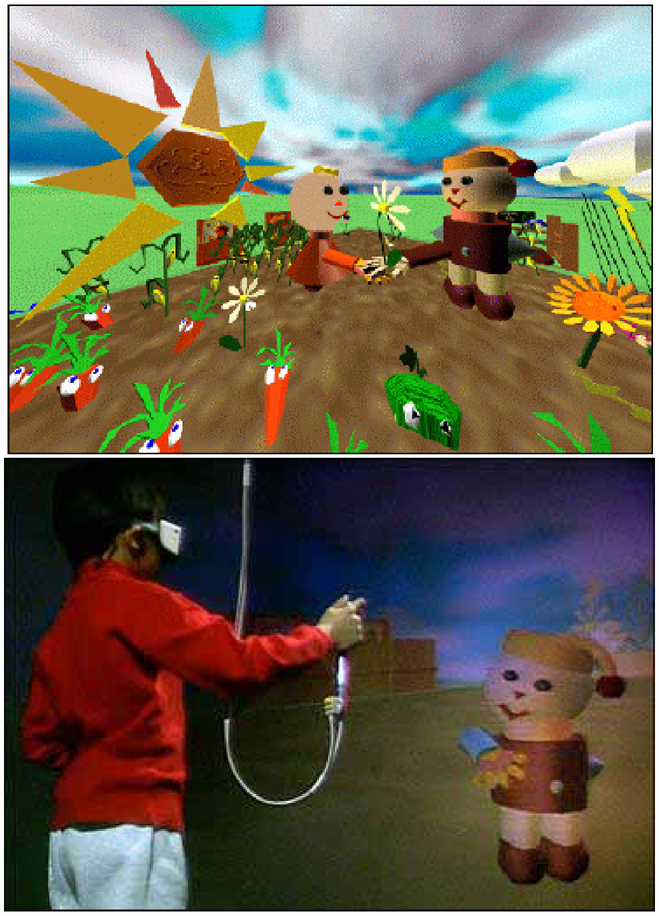


Figure 2. NICE: a narrative immersive collaborative environment for education. The top scene shows an avatar handing a flower to another avatar in the NICE garden. Below is an image of a child interacting with an avatar in the CAVE.

continues to evolve; the plants in the garden keep growing and the autonomous creatures that inhabit the island remain active.

Interactions with the NICE garden are not limited to users with VR hardware. The garden in NICE can be experienced either by entering VR, a basic WWW browser (http://www.ice.eecs.uic.edu/~nice), a VRML2 browser, or in a Java applet. Participants using a mouse can interact with participants using VR hardware where the desktop user's mouse position is used to position an avatar in the 3D virtual world, and the bodies of the VR users are used to position 2D icons on the desktop screen. This kind of scalability will be important for increasing the breadth of possible collaborations.

1.3 Collaborative Scientific Visualization

A typical scenario in collaborative scientific visualization is for a small group of scientists that are remotely located, to enter a virtual environment to discuss a data set that is being visualized. This data set may originate from a database or may be computing simultaneously on a supercomputer, in which case the virtual environment can be used to steer the computation(11). The importance of collaboration in this environment is not so much in allowing the remote participants to perform different tasks simultaneously as it is to allow them to offer their different opinions over what is observed in the visualization. The default assumption is that all the participants should share a homogeneous view. However for a complex data set that spans numerous dimensions, it may be more useful to partition the dimensions so that different virtual environments observe different dimensions during the simulation.

Argonne National Laboratory (ANL) in collaboration with Nalco Fuel Tech have built an immersive interactive engineering tool for designing pollution control systems for commercial boilers and incinerators(12). Using ANL's CAVEcomm library multiple CAVEs could synchronously connect with an IBM SP supercomputer to steer the interactive simulation of flue gas flow in the boiler. Control of the simulation was strictly via turn-taking. One participant could initiate the flow from one viewing location while another participant could simultaneously view the flow in a different chamber of the boiler. Participants could communicate with one another via a conference telephone call.

As with most teleimmersive applications, this system is only in a prototypical phase. Additional capabilities that may be useful in enhancing work in the environment include:

1. Discovery Recording - the ability to annotate (perhaps using voice recording) to mark points of interest in the data set- storing the annotation, and the state of the environment when the "snapshot" was taken. This will allow the engineers to return to the time of the event and re-observe it.

2. Storing Computed or Raw Data Sets - typically the data generated by a simulation or gathered from data-gathering devises are too large to fit into the physical memory of the computer performing the visualization. In this case some scheme of hierarchically storing this data is needed to allow querying for smaller subsets of the data for visualization.

CHAPTER 2

THE PARTICULAR REQUIREMENTS OF TELEIMMERSION

The scenarios described in the previous chapter illustrate the broad spectrum of human-factors, graphics, networking, and database requirements that are needed to support teleimmersion. These requirements are elaborated in this chapter.

2.1 Avatars

When collaborating in VR, virtual representations are needed to uniquely identify each participant. The popular default assumption for representing avatars in VR is to place texture-mapped, live video images on the head of a three-dimensional model of a human. The problems with this scheme are manifold: the bandwidth required to transmit video may be too high to "waste" on sending facial images; most VR experiences require the participant to wear some form of head gear which will typically occlude most of the participant's face making video images of faces impractical; attaching a video camera and light source, however small, in an optimal position to capture images significantly increases the encumberances already inherent in the VR gear.

The elaborateness of the avatar should vary with the task being performed. Hence it is important to identify the minimum elements of representation needed to afford recognizability and to convey non-verbal information such as body language and gesture. In our experience we have found a minimum of head position and orientation, body direction, and hand position and orientation to be adequate for many teleimmersive tasks. To afford recognizability, we have found it easier to distinguish avatars based on geometry rather than color. Hence the commonly used, homogeneously shaped avatars with varying colors and overlaid name tags, do not make good avatars.

To support the minimal avatar, a bandwidth of approximately 12Kbits/sec[1] (at 30 frames per second) is needed. Theoretically this implies that 10 avatars can be supported over a 128Kbits/sec ISDN connection. In practice however, experience has shown that it is able to support a maximum of four avatars with an average latency of 60ms using UDP as the transmission protocol. Although this is not a scalable solution, it is a cost effective means of transmitting VR avatar data with the quality of service of a dedicated connection.

2.2 Suitable Interfaces for Collaborative Manipulation and Visualization

High-level virtual interfaces must be developed to allow collaborative manipulation of shared objects. In addition, these manipulation tools require some form of transparent locking to occur so that consistency is maintained across all the virtual environments sharing the virtual space. The goal is to provide mechanisms for acquiring distributed locks (possibly through predictive means) so that the user does not realize that locks have had to be acquired before objects could be manipulated. This is particularly important over high latency networks where there might be

[1]This includes the avatar's head and one hand's position (x,y,z), orientation (θ_x,θ_y,θ_z), and body angle about the Y axis represented each as 4-byte floating pointer numbers.

noticeable delay between the time when a user physically picks up an object (and hence attempts a lock on it,) to the time when the VR system confirms the lock on the object. Lag similar to this has been shown to significantly degrade human performance in a VR environment(13). Previous work in this area has shown that for coordinated VR tasks involving two expert VR users, performance begins to degrade when network latency increases above 200ms(14). Other research has found acceptable latencies to be much lower (100ms)(7). The acceptable latency is expected to be lower for inexperienced users and for coordinated tasks involving very fine manipulation of shared objects. In the latter situation tracker inaccuracy will also begin to affect human performance.

2.3 Audio/Video Teleconferencing

Audio (voice telephony) is one of the most important channels to provide in a collaborative experience(15; 16). It has been shown that latencies of greater than 200ms will result in degradations in conversion(17). As the latencies continue to increase the amount of time spent in confirming conversion increases, and the amount of useful information being conveyed in the conversation decreases. Video conferencing is useful in instances where it is important for the participants to see each other face to face for negotiation tasks(18; 19; 20). In traditional conference-room-style video conferencing, video provides a means to convey a sense of co-presence(21). In VR however co-presence is directly created through the use of avatars and hence video may play a less significant role in the collaboration.

2.4 Synchronous and Asynchronous Collaboration

The main focus of most teleimmersive applications has previously been on synchronous collaboration. That is, all participants are working together in the environment at the same time. However in trans-global collaborations the timezone differences make routine synchronous collaboration highly inconvenient. In this case it is important to also provide a means for distributed groups to work asynchronously in a shared virtual space. The support of asynchrony will require the use of distributed databases to maintain the states between the remote sites.

2.5 Persistence in Collaborative Virtual Reality

Persistence in Collaborative Virtual Reality describes the extent to which the virtual environment exists after all participants have left the environment. Persistence can be divided into three major classes: participatory persistence, state persistence, and continuous persistence.

2.5.1 Participatory Persistence

This is persistence in which the VE only exists in the brief amount of time that participants are in it. When all participants leave, the environment is extinguished with no record of the state of the environment before it was extinguished. When the environment is started at a later time, it always begins at the beginning. Most virtual environments are still only participatory persistent.

2.5.2 State Persistence

This is where the state of the virtual environment may be saved at any given time to be recalled later. Either intermittent snapshots can be created or entire collaborative experiences can be recorded for later review.

In a scientific visualization environment that involves simulations that are running on supercomputers a recording should include either the entire state of the virtual environment as well as the state and output of the simulation, or the state of the virtual environment and only the geometric representation of the simulation. The advantage of the latter is that it simplifies the mechanism for re-play. Re-play will only require a rendering of the geometry which can easily be encapsulated to work even in external viewers such as VRML2 browsers. However the disadvantage is that the geometry data itself cannot be re-used to further query the output of the simulation.

On the otherhand the advantage of saving the state of the environment and the simulation is that during re-play the participant can choose to dynamically re-involve the supercomputer. This is motivated by the following:

To ensure accuracy in computational science simulations, the simulation steps are kept relatively small. However the computed results are collected at every n time steps due to, surprisingly, disk space limitations (the output can occupy between several hundred megabytes to many gigabytes)(11). If by viewing the results a feature of interest is found, the scientist would normally re-execute the simulation from the beginning but only begin recording the output in the region of the feature and at each time step rather than at every n time steps. In this scenario persistence may be used to offer some assistance in reducing the amount of re-computation time. State Persistence may be invoked during the initial course-grained recording where, instead of simply recording the output, the states of the computation are also recorded. When the region of interest has been isolated, rather than returning computation to the beginning as is typical, the state of the computation can be retrieved from the persistent database and computation can be resumed from that point. Recording can then resume at a finer granularity.

In general, as part of the recording of persistent experiences it may be useful to also record the actions of the avatars so that on re-play they may be re-positioned in the scene to serve as reminders of which particular area of the visualization was being observed or manipulated at the time of the recording. In fact it may actually seem rather unnatural to watch an event transpire without being able to see the effector of the event. One could also imagine that the re-play procedure may also be recursively recorded so that a participant could observe him/herself observing him/herself. It is not entirely clear if this capability is of any value but the idea is at least somewhat intriguing.

Finally, one important component of being able to record a virtual experience is to be able to perform (temporal) queries on the recording. Examples of queries might be: "Show me when this part was modified," "Show me when Max walked by here," "Show me when this data set exceeded this threshold," "Show me all the objects Max modified."

2.5.3 Continuous Persistence

This is where the state of the virtual environment remains extant even when all the participants have left. Hence when participants re-enter the environment the state of the world may have changed. Such environments are liken to MUDs (Multiuser Domains/Dungeons) which by their popularity, have shown to encourage the spontaneous use of collaborative environments.

Although this may seem to be an extravagant use of computing power, it is anticipated that in future generations of teleimmersive environments the notion of persistence is merely an extension of the existing idea of the operating system or the WWW server. These are essentially, already continuously persistent environments.

2.6 Flexible Support of Various Data Characteristics

The design of Teleimmersion systems is affected by two interrelated factors: the characteristics of the data being distributed and the distribution scheme employed. The four attributes that characterize teleimmersion data that most greatly affect the mode of transmission, management and storage of teleimmersion data, are: quality of service, data size, persistence and queueing.

2.7 Network Quality of Service

For closely coordinated work in teleimmersion, minimum levels of network bandwidth, latency and jitter are desirable. In addition, both reliable and unreliable protocols of unicast, broadcast and multicast transmission are needed to optimally transport different classes of teleimmersion data (3D tracker data, state information, streamed audio/video feeds, geometric models, large scientific data sets.)

Unreliable protocols are suitable for the transmission of tracker data because: 1. the loss of a packet of tracker data is usually followed shortly afterwards by newer ones, and 2. unreliable protocols have a lower latency and utilize lower bandwidth than reliable protocols.

Multicasting has the additional benefit that clients that subscribe to a multicast group need only send one message to the group, rather than having to send the same message individually to each participant in the collaboration. The multicast protocol will automatically propagate the single message to all the other subscribers. The main disadvantage however is that multicast is based on unreliable UDP. Work however, is currently underway in developing reliable multicast protocols (22). Reliable transmission is important in teleimmersion for the delivery of accurate state information as well as models and scientific data sets. Here the loss of a packet could produce an unwanted artifact in the visualization that is not representative of the original data set.

A flexible solution to networking for teleimmersion should include both reliable and unreliable forms of transmission. However it is interesting to note that only a few provide both capabilities simultaneously(23; 9). This is perhaps due to the following reasons: First, the main concentration of VR libraries in the past has been in providing tools to allow programmers to quickly build interactive non-collaborative VR environments (e.g MRToolkit(24)). Support for collaboration was generally an after-thought and hence reliable TCP is used as the default, safe and generic solution. Secondly, most teleimmersion implementations are still experimental technologies undergoing significant change. For example DIVE(25) initially used a transaction-oriented, object-oriented database called ISIS and a reliable TCP connection to synchronize all state information in the TIE. They are now using a peer-to-peer connection with a replicated database that synchronizes data via a reliable multicast connection. Finally, the implementations may be highly customized for specific problem domains. For example NPSNET(7) uses multicasting to deliver information for military simulations. Other researchers have attempted to extend the underlying DIS protocol to allow the delivery of "non-ballistic" information. But because it uses an unreliable protocol additional mechanisms for retransmitting packets had to be devised. In addition, since the notion of military weapons are directly embedded in the specification of the protocol it does not serve as a generic protocol for non-military simulations such as collaborative engineering or scientific visualization.

2.7.1 Data Size

There are essentially three categories of teleimmersion data sizes: small-event, medium-atomic, and large-segmented. These divisions are created because they affect the manner in which they are optimally transmitted and manipulated.

- **Small-Event data** are data such as unreliable tracker data, and reliable state and event data. These typically require priority transmission with low latency.

- **Medium-Atomic data** are data that are small enough to fit in the physical memory of the client because it must be processed as one atomic "chunk." Examples of these are 3D geometries representing individual objects in the VR scene.
- **Large-Segmented data** are data that are too large to fit in the physical memory of the client and hence can only be accessed in smaller segments. Large scientific data sets and long pre-digitized video streams fit this category. These data sets usually need to be "abstracted-down" first before they can be visualized, as the amount of data that can potentially be visualized can easily exceed the graphics rendering capabilities of the VR system.

2.7.2 Queued/Unqueued Data

Data that are sent to clients or servers, regardless of whether they are stored in a database or not, need to be either queued or unqueued. For example, world state information may be unqueued since only the latest information is necessary. Queued data are data which must all arrive at a client or server in order. This implies the use of a reliable protocol. There are however instances where a queued, unreliable protocol may still be useful- specifically for audio conferencing, long, unreliable data streams are transmitted to all participating clients.

2.7.3 Persistent/Transient Data

Persistent data characterizes data that needs to be stored in a database or file system for later use. This data remains in the database after all the clients leave the TIE. All state data that is crucial to the resumption of a client in a teleimmersive session must be persistent. Models and scientific datasets that will be loaded into TIE are also prime candidates for database storage.

Transient data are data that are not stored in a database. An example of this kind of data are command messages that might be sent between clients to effect events or audio/video data streams. An exception to this definition is when transient data is stored in a database to allow re-play of events at a later time. In this case the data is more accurately characterized as persistent rather than transient.

However there is more to persistence than simply the storage of data. The storage requirements of various types of Teleimmersion data are illustrated in Figure 3. This table maps the networking and database requirements for three modes of teleimmersive interaction (unrecorded interaction, periodic snapshots, continuous recording) and six categories of Teleimmersion data (raw scientific data, derived data, avatar data, virtual state and meta data, three dimensional models, and video conferencing data.)

Note for example that the database requirements for recording avatar data is different for periodic snapshots as it is for continuous recording. For continuous recording, database integrity can be sacrificed for throughput to ensure that all realtime changes to the environment are captured. For periodic snapshots it may be important that the snapshots are made reliably so that on recall they represent the entire environment consistently. These and other differing database requirements suggest that no single database system will support all the needs of teleimmersive applications. Instead a number of databases with varying capabilities should be gathered and unified under a single consistent interface which will allow the teleimmersive application to negotiate the kind of "database quality of service" needed by the application.

2.8 Scalable and Flexible Topological Construction

No single interconnection of distributed resources will perform optimally for all teleimmersive applications. The number of participants expected to work in the environment, the amount and form of the data being shared, the geographic distance, and the intervening networks connecting participants, have profound effects on the design of a suitable distributed topology. Systems that are designed to scale well with respect to connectivity (connection scalability) typically must sacrifice strong data consistency. Most currently existing systems prioritize connection scalability over data scalability (ability of TIEs to handle enormous amounts of data.)

Data scalability is of greater importance to the development of engineering and scientific applications than connection scalability. Data sets in these problem domains are typically enormous in size however the number of people simultaneously collaborating is unlikely to exceed 6 or 7.

The three main classes of distributed topologies used in teleimmersion include: replicated homogeneous, shared centralized, and shared distributed(7). These are described below.

2.8.1 Replicated Homogeneous

Replicated Homogeneous topologies are classical of military VR simulations (as in SIMNET, NPSNET, DIS)(7). In such topologies each client holds a completely replicated database of the shared environment and state information is shared by broadcasting messages to all participat-

	Volume	Rate of Change	Persistent	Complexity of Data	Database integrity	Type of DB Transaction	Network QoS Requirements
No Recording							
Raw scientific data set	L	SD	Y	C	R	WO,SA,SU	R
Derived data	MA - L	SCD	Y & N	V	R	SA,SU	R
Avatar data	S	RCD	N	S	-	-	U,L,J
Virtual world state & meta data	S	SCD/RCD	N	V	-	-	R,L,J
3D models	MA - L	SD	Y	V	R	SU	R
Video conferencing data	MA	RCD	N	V	-	-	U,B,L,J
Continuous Recording							
Raw scientfic data set	R	SD	Y	C	-	-	R
Derived Data	MA	SCD	Y	V	R	SA	R
Avatar data	S	RCD	Y	S	D	RA	U,L,J
Virtual world state & meta data	S	IRCD	Y	V	D - R	RA - SA	R,L,J
3D models	R	SD	Y	V	-	-	R
Video conferencing data	MA	RCD	Y	V	D	RA	U,B,L,J
Intermittent Snapshots							
Raw scientfic data set	R	SD	Y	C	-	-	R
Derived Data	MA	SCD	Y	V	R	SU	R
Avatar data	S	SCD	Y	S	R	SU	U,L,J
Virtual world state & meta data	S	SCD	Y	V	-	SU	R,L,J
3D models	R	SD	Y	V	-	-	R
Video conferencing data	MA	SCD	Y	V	R	SU	U,B,L,J
Derived = data derived from	L=Large	SD=Static Data		C=Complex	R=Required	SA=Safe Appends	B=Bandwidth guaran. desired
raw data sets.	MA= Medium Atomic	SCD=Slow Changing Data		V=Varies	D=Desired	SU=Safe Updates	R=Reliable
Meta data = miscellaneous data	S = Small	RCD=Rapid Changing Data		S=Simple		RU=Rapid Updates	U=Unreliable
coordinating all the other data	R=Referenced	IRCD=Intermittent RCD				WO=Write Once	L=Latency guarantee desired
						RA=Rapid Appends	J=Jitter guarantee desired

Figure 3. Mapping the networking and database requirements for three modes of teleimmersive interaction (realtime interaction, state persistence, continuous persistence) in six categories of Teleimmersion data (raw scientific data, derived data, avatar data, virtual state and meta data, three dimensional models, and video conferencing data.)

ing clients. This system has no centralized control whatsoever, hence any new client joining a session must wait and gather state information about the world that is broadcasted by the other clients.

2.8.2 Shared Centralized

In this approach all shared data is stored at a central server. The main advantage of this scheme is that it greatly simplifies the management of multiple clients, especially in situations requiring strict concurrency control. However, its role as an intermediary for the delivery of data can impose an additional lag in the system. Another disadvantage is that if the central server fails none of the connected clients can interact with each other. Despite these disadvantages, this architecture is still useful for supporting small groups of collaborators.

2.8.3 Shared Distributed with Peer-to-peer Updates

This approach simulates a wide-area shared memory structure (25; 24; 23; 26) in which objects that are instantiated at one site are automatically replicated at all the remote sites. This logical abstraction simplifies the application development at the cost of performance. Typically in these implementations, a newly connected client must form point-to-point connections with all the participating clients. Hence for n participants the number of connections required is $n(n-1)/2$. In addition if the environment involves the sharing of enormous scientific data sets, the data set will be fully replicated at every site. Unless the data sharing policy is modified to account for large datasets this scheme will not be scalable.

2.8.4 Shared Distributed using Client-server Subgrouping

This topology distributes the database amongst multiple servers. Clients connect to the appropriate server as needed. A classic approach is to bind the servers to unique multicast addresses. Clients then subscribe to different multicast addresses to listen to broadcasts from the servers(27; 28). This is a particularly effective way to handle large numbers of connected clients distributed over a wide virtual space. Each geographic region of the virtual space can be maintained by a separate server. The servers share the load of sustaining the state of the virtual world by handling only the subset of the connected clients that are in their geographic region.

2.9 Application Specific Servers

These are unlike traditional networking and database servers in that they do not simply store and forward data. Application specific servers in VR also possess semi-graphical capabilities as they may need a local representation of the virtual space for their operation. For example, an application specific server simulating the movement of autonomous agents through a virtual landscape may also use the same graphical routines that model and visualize the terrain to perform operations such as collision detection.

2.10 Interoperability with Heterogeneous Systems

The varying domains in which teleimmersion is applied requires connectivity between heterogeneous resources such as external databases, supercomputers, desktop workstations, and miscellaneous VR systems. For example, Argonne's incinerator simulator connects the CAVE

VR system to an IBM SP supercomputer. The supercomputer performs the computation while the CAVE visualizes the results. In NICE, the system allows CAVEs, ImmersaDesks, desktop workstations, WWW browsers and Java programs to all collaborate simultaneously.

CHAPTER 3

THE APPROACH

In light of all the complex, interacting aspects of computer graphics, networking, databases, and human-factors that come into play in teleimmersion, developing teleimmersive applications can be a daunting task. The temptation and common mistake, made by application developers that are building teleimmersive applications for the first time, is that they will first build a non-collaborative application and then attempt to retro-fit it for teleimmersive capabilities. It is in fact more difficult to retro-fit an application for teleimmersive capabilities than to introduce them early in the design phases of the application. Hence it is important to provide tools that will encourage application developers to envision teleimmersive scenarios at a high-level so that they can determine how such capabilities would be most useful in their own applications. However a high-level set of tools does not offer much help for those trying to retro-fit existing applications. A high level library of well integrated tools often assumes a specific software-design methodology. This methodology may be incompatible with the software that is being retro-fitted. For example a high-level library such as DIVE is a good system for rapidly constructing new teleimmersive applications, but it cannot be used for adding teleimmersive capabilities to an existing Performer CAVE application. The mechanisms for graphics rendering in DIVE and Performer are incompatible and DIVE is not modularized enough to allow arbitrary use of its individual components.

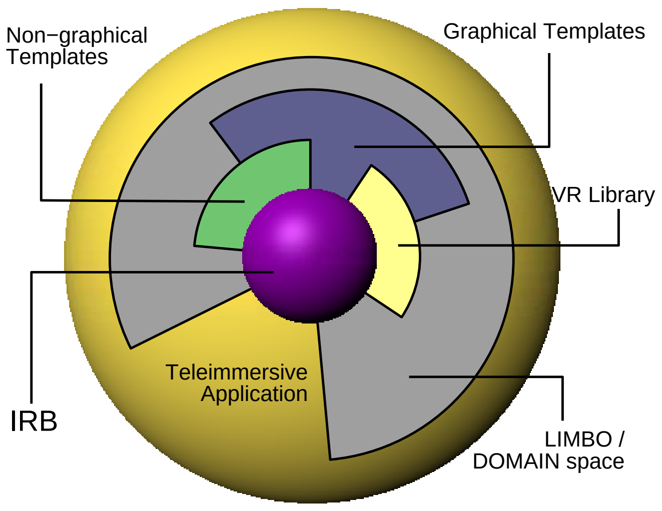


Figure 4. Software infrastructure of CAVERNsoft.

To address this issue this dissertation proposes a software infrastructure (called CAVERNsoft) that will support both the rapid creation of new teleimmersive applications, and the retro-fitting of previously non-collaborative VR applications with teleimmersive capabilities.

3.1 CAVERNsoft

CAVERNsoft, shown in Figure 4, consists of a central structure called the Information Resource Broker (or IRB) surrounded by layers of support software. Although these layers appear to increasingly hide the lower layers from the main application they are in fact accessible at every level. The lower-levels facilitate the construction of new components, and the retro-fitting of existing applications. The higher-levels facilitate the rapid development of new teleimmersive applications.

The IRB is a relatively low-level merging of networking and database capabilities that is completely separate from graphics. Hence the basic IRB core can be placed in any software application regardless of whether it possesses graphics capabilities. This allows graphical applications to communicate with non-graphical applications and it also allows existing non-collaborative applications to possess networking capabilities with minimal disturbance to their existing mechanisms for rendering graphics.

At a layer above the IRB are still non-graphical template libraries that support such things as: base classes for the coordination of avatars, and audio and video data compression algorithms. On top of this layer is a higher level layer that consists of graphical versions of the previous layer. For example OpenGL, Performer, and Video avatar templates. These higher level templates can then be gathered into even higher level fully functional teleimmersion spaces called **LIMBO** spaces.

LIMBO spaces will provide varying degrees of avatar rendering and recording; model importing, distribution, manipulation and version control; and audio/video teleconferencing. These individual elements will be integrated in a manner that is guided by research in human-factors in cooperative work situations. Using a basic LIMBO space collaborators can begin working in the virtual space immediately. They may enter the space with an avatar of their choosing and import 3D models (perhaps of car designs, or scientific data-sets, etc) into the space. The space will ensure proper distribution of the model to all the other remote participants. Once the objects are distributed the participants may collectively modify them. In addition, application developers may use the well documented source code of the LIMBO space to jumpstart the development of their own domain-specific teleimmersive applications.

As more domain-specific applications are developed a growing library of CAVERNsoft-based reusable components (such as collaborative visualization tools) will emerge. These can be added to the library of existing templates and may be gathered to build DOMAIN spaces that are specializations of LIMBO spaces. This will allow, for example a designer, to build a teleimmersive design application by starting with an existing DOMAIN space that is equipped with collaborative tools specifically for collaborative design, rather than starting from the basic LIMBO space.

3.2 The Information Resource Broker

The full development of CAVERNsoft is beyond the scope of this dissertation. This dissertation will focus on only one aspect of CAVERNsoft- the central core that will support data distribution between CAVERNsoft applications. That is, this dissertation proposes and implements a subset of the IRB.

The Information Resource Broker (IRB) is the nucleus of all CAVERN-based client and server applications. The ultimate goal of the IRB is to place powerful networking and database tools, that embody the expertise of networking and database researchers, at the finger tips of application developers. These tools should be presented with a unified interface so that the programmer does not have to learn separate networking and database models of operation. This problem however is not solved by simply employing an existing distributed shared memory system or distributed database. The realtime requirements of teleimmersion make these more-reliable, and consequently, constraining, solutions unsuitable. A suitable architecture for Teleimmersion should: facilitate the rapid construction of arbitrary distributed topologies; provide support for various networking protocols (including reliable and unreliable, unicast, broadcast and multicast) and quality of service capabilities; provide facilities for supporting concurrent programming (both a message-passing and a distributed shared memory model); and provide support for persistence- where small-event, medium-atomic and large-segmented data can be seamlessly managed.

The IRB supports these requirements with an architecture that is a hybrid of a realtime networking library, a distributed shared memory system and a distributed database. It offers a unified high-level interface to these capabilities while still providing the necessary low-level control necessary to manage realtime data.

A client application is built by using an IRB interface (IRBi) which, on invocation, will spawn the client's "personal" IRB. This IRB is used to cache data retrieved from other IRBs during the operation of the client. An application-specific server is similarly built using the IRBi. Hence there is little differentiation between a client and a server (Figure 5.) Using the IRBi a client can arbitrarily form a connection, after having acquired the proper permissions, with any other client or server to access its resources. The IRBi will communicate the request to the client's personal IRB which will then communicate with the remote client's or server's IRB. It is the IRBs' responsibility to negotiate the networking and database services requested by the client/server applications. This form of flexibility and symmetry will allow all

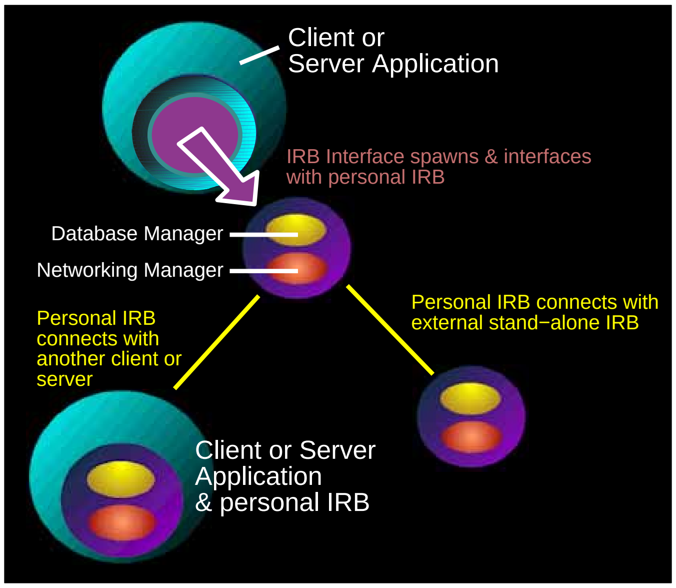


Figure 5. Clients/Servers use the IRB interface to spawn personal IRBs with which to communicate with other clients/servers or standalone IRBs.

of the main teleimmersive topologies to be quickly constructed. Figure 6a. shows IRB-based clients with possibly fully replicated databases sharing updates via a multicast group (as in DIS/NPSNET(7).) Figure 6b. shows the use of IRBs in a shared, centralized database (as in CALVIN and NICE.) Figure 6c. shows IRB-based clients in a fully connected configuration to support a shared, distributed database with peer-to-peer updates (as in MRToolkit(24).) Finally Figure 6d. shows IRB-based clients and servers that are connected to form a shared, distributed client-server database (as in SPLINE(27).) The clients may arbitrarily connect to any of the servers using any desired communications protocol to retrieve information. Since

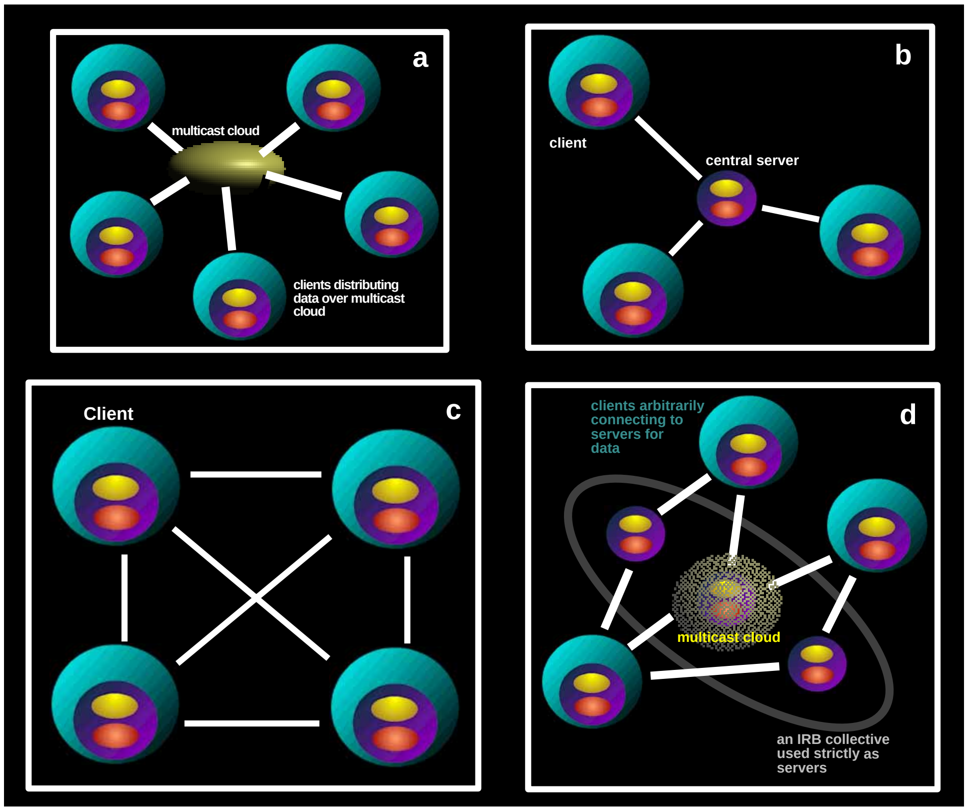

a
multicast cloud
clients distributing data over multicast cloud
b
client
central server
c
Client
d
clients arbitrarily connecting to servers for data
multicast cloud
an IRB collective used strictly as servers

Figure 6. Use of IRBs to construct all the major classes of Teleimmersion topologies. (a) Fully replicated databases sharing updates via a multicast group/cloud. (b) IRB clients connected to a shared centralized database (also an IRB.) (c) IRB clients in a fully connected configuration to support a shared, distributed database with peer-to-peer updates. (d) IRB clients and servers connected to form a shared, distributed client-server database.

there is no distinction between a client or a server, an IRB-based program may be a client running on a supercomputer, or a server interfacing with a large database of scientific data.

3.3 The IRB Interface

The IRB interface (IRBi) is the client and server's interface to the IRB. The IRBi provides the application with a handle to a personal IRB that the application can use to activate dynamic connections with remote IRBs. A client wishing to share information between its personal IRB and a remote IRB begins by first creating a communication channel and declaring its communication properties. Then any number of local and remote keys may be linked over the channel (Figure 7.) A key is a handle to an arena of memory that can be committed to the IRB's persistent store. Keys are uniquely identified across all IRBs and can be hierarchically organized much like a UNIX directory structure. Each local key may be linked only once to a remote key on a remote IRB. That is the same key cannot be linked twice to the same IRB. However each local key can initiate or accept multiple linkages to and from other remote keys on different IRBs. The application is generally unaware of these additional linkages as the personal IRB transparently manages data sharing with the remote subscribers. The application is only aware of the linkages that it has explicitly made. When keys are linked, any modifications made to one key will automatically be propagated to all the other linked keys based on the individual link properties.

3.3.1 Channel Properties

Channel properties allow clients to specify the networking service desired for data delivery. Clients may specify reliable TCP, or unreliable UDP and multicast. Large packets delivered

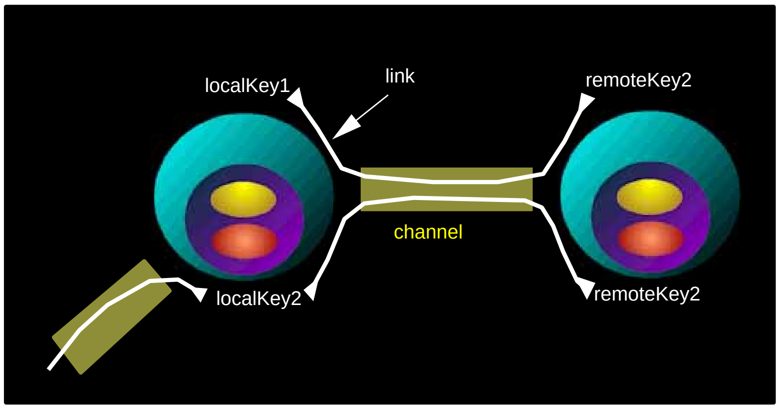


Figure 7. Example of two local keys linked to remote keys on remote IRBs.

over unreliable channels will automatically be fragmented at the source and reconstructed at the destination. If any fragment is lost while in transit the entire packet is rejected.

In addition to connection reliability clients may specify Quality of Service (QoS) requirements. Hence they are able to declare the desired bandwidth, latency, and jitter of the data stream. The personal IRB will attempt to obtain the desired level of QoS from the remote IRB, but if it fails, the client may at any time negotiate for a lower QoS. As in RSVP(29) client-initiated QoS is used so that the client can specify the amount of data it can handle from the remote IRB.

3.3.2 Link Properties

Link properties allow clients to specify the actions taken when local and remote keys are linked. This includes being able to choose between active and passive updates and being able to select the initial and subsequent synchronization behavior. Hence it allows the IRB to support both write-update and write-invalidate coherence protocols as in a distributed shared memory.

In most teleimmersive applications, world state information consisting of a few tens of bytes are *actively* distributed. That is, the moment a new value is generated it is automatically propagated to all the subscribers of the data. *Passive* updates occur only on subscriber request and usually involves a comparison of local and remote timestamps before transmission. For example, passive updates are typically used to download large volumes of 3D model data. Caching data and comparing their timestamps helps to reduce the need to redundantly download the same data set.

The initial synchronization behavior determines how the local and remote keys should be synchronized when the links are first formed. That is, clients are able to choose to synchronize automatically based on the keys' timestamps. That is the older key will be updated with information from the newer key. However the client may also choose to force synchronization from the local key to the remote key, and vice versa, regardless of timestamp. Of course clients may choose to perform no initial synchronization at all.

Subsequent synchronization behavior specifies the manner in which data is synchronized when local or remote updates to keys occur. The same options as for initial synchronization apply.

The default link property is to use active updates with automatic initial and subsequent synchronization.

3.3.3 Key Properties

Keys may be defined at a client's personal IRB or at a remote IRB provided the client has the necessary permissions. Keys may either be transient or persistent. Persistent keys are keys that will be stored in the IRB's datastore so that when a client or server re-launches, the data will still be retrievable by specifying the same key identifier. Clients determine whether a key is to persist by asking the IRB to perform a commit operation on the data. In addition simple locking functions should be provided to allow clients to lock local or remote keys (hence permitting entry consistency.) Locking calls are non-blocking to prevent realtime applications from stalling when attempting to acquire locks on keys. Instead the locking call accepts a user-specified callback function that will be called when a lock has been acquired or when any relevant event pertaining to the lock occurs.

3.3.4 Asynchronous Triggering of Events

Although distributed shared memory (DSM) systems have historically been shown to be easier to program than message passing systems(30), Teleimmersion requires that the DSM model be enhanced with some message passing capabilities. Specifically it should be enhanced with the ability to asynchronously generate events in the application. Many events may arise during the course of distributing data between clients and servers. The client/server may need to be notified so that appropriate actions may be taken in response to these events. It is inefficient for realtime VR applications to continuously poll for such conditions. IRB-based programs provide the IRBi with callback functions that the IRBi may call when the event arises.

Examples of events include:

- **New Incoming Data Event**

  This event occurs when a key receives a new piece of data. For example a key could be subscribing to avatar state information (position and orientation of the avatar's head). When this information changes, a callback can be invoked to make the corresponding changes to the graphical representations of the avatars in the virtual world.

- **IRB Connection Broken Event**

  When a connection to an IRB has been broken (possibly due to a crash) clients will continue to function by accessing local versions of the subscribed data. The personal IRB may then periodically attempt to re-establish connection with the remote IRB or choose another client or server to take the place of the "broken" IRB. It is the responsibility of the application-specific IRBs to determine the policies for such situations.

- **QoS Deviation Event**

  This event occurs when the negotiated QoS falls below contracted levels. For example, if the latency negotiated for a stream of tracker data falls below acceptable limits, the VR client can be warned so that perhaps interpolative techniques such as dead-reckoning can be activated to reduce the impact of the increased latency. Alternatively the client can re-negotiate a different QoS, perhaps one involving a lowering of the bandwidth (by compression of data) in order to maintain the desired latency.

3.3.5 Direct Connection Interface

In addition to the many automatic networking capabilities provided by IRBs the IRBi must still support direct access to low-level socket TCP, UDP, multicast interfaces so that connectivity with legacy systems (such as WWW servers) can be supported. However CAVERNsoft adds value to the basic socket-level interfaces by providing automatic mechanisms for accepting new connections, and making asynchronous data-driven calls to user-defined callbacks.

3.3.6 Supplementary Concurrent Processing Facilities

Most of the networking and database operations performed in the IRB are executed concurrently and, if a multiprocessor system is available, in parallel with the VR system. It is therefore necessary to provide basic concurrency control primitives such as mutual exclusion and condition variables, that are compatible with the IRB. These may be implemented as macro definitions on top of the underlying threads library used by the IRB (for example POSIX threads.)

3.3.7 Recording Keys

The IRBi should allow the clients to declare keys that hold recordings of groups of keys. This facility is necessary to support State Persistence in VR.

In these recordings close synchronization of remote system clocks is not absolutely necessary as recording is always made from one point of view (i.e. from a virtual camera) and hence it is the point of view's time reference that all relevant information is recorded.

Recordings may consist of time stamping and storing every change in value that occurs at a key and recording the state of all the keys at wide intervals. The former is needed to track the gradual changes in the virtual environment over time. The latter is needed to establish checkpoints so that the recordings may be fast-forwarded or rewound without having to compute every successive state that led to the fast-forwarded/rewound location.

On playback the recordings will populate the appropriate keys and, if desired, trigger client callbacks. In some instances it is useful to be able to playback only a subset of the recorded keys. This will allow the user to observe smaller subsets of events that occur in the VR environment. For example the Virtual Director(31) (a VR application that allows users to record the path of a virtual camera through a virtual environment) allows playback of recordings of camera positions in each of the three X,Y,Z axes so that each of the paths in the axes can be edited independently.

Finally to synchronize the playback of experiences across multiple virtual environments each environment must constantly broadcast their frame-rate. This ensures that faster VR systems do not overtake slower systems while rendering the virtual imagery.

In order to support these and many other recording capabilities in Teleimmersion, the IRB must adopt a notion similar to quality of service for networks. That is, the IRB needs a mechanism for the client/server program to be able to negotiate the kinds of data storage throughput and integrity needed for its particular application (as indicated in section 2.7.3.)

CHAPTER 4

IMPLEMENTATION OF THE IRB

In this chapter a detailed description of the implementation of a prototype IRB will be presented. This prototype does not implement all the capabilities of the IRB, it only implements the main capabilities that define the IRB. Specifically the features include:

1. The ability to arbitrarily define local and remote keys on networked IRBs.
2. The ability to create multiple communications channels between arbitrary IRBs using both reliable (TCP) and unreliable (UDP, multicast) protocols.
3. The ability to link keys across communication channels and have them automatically propagate data. Clients are able to select different initial and subsequent synchronization mechanisms as well as active and passive updates.
4. The ability to trigger on key events and connection events so that the IRBs can inform client-applications of new data or broken connections.
5. A prototype persistent key datastore has been implemented. This preliminary implementation caches all accessed persistent data in main memory and maps the key hierarchy to a UNIX directory and file system, and hence it supports small-event and medium-atomic data.

The initial prototype was implemented to run on any O2, Octane and Onyx2 generation of Silicon Graphics computers. This was chosen primarily because they are currently the main platforms for the development of high-end teleimmersive applications. However, a port to other UNIX-based platforms should not be difficult as the IRB contains no graphics capabilities.

The IRB's underlying networking is supported by Nexus(32). Nexus is an efficient multi-threaded communications library developed by Argonne National Laboratory to connect client applications with remote supercomputing resources.

Nexus supports five basic abstractions: Nodes, Contexts, Threads, Communication Start-points and Endpoints, and Remote Service Requests. Nodes refer to computational resources such as workstations and processors on supercomputers. Contexts are Nexus-based programs or forked processes that run on nodes. Threads are concurrent, light-weight "sub-processes" that share the same address space as a UNIX forked process. Communication Startpoints and Endpoints are created between communicating contexts. Remote service requests are essentially remote procedure calls (without a synchronous reply mechanism) that are initiated from a startpoint to call a remote function at an endpoint. When a request arrives at the endpoint a thread is created to process the remote function. Many startpoints may be bound to a single endpoint and hence an endpoint may handle requests for many remote contexts. This model of threaded remote procedure call without a reply facility was chosen by the designers of Nexus to maximize asynchronous communication and parallelism so that programs do not have to block while waiting for replies from remote calls.

Future releases of Nexus will also provide networking quality of service capabilities. The CAVERNsoft programmer can then access these capabilities through the IRB's API. Hence as new networking capabilities are provided by Nexus, these same capabilities will be made available to the IRB.

4.1 Overall Structure of the Information Resource Broker

The IRB consists mainly of three components: the Communication Manager, the Active Key Manager and the Persistent Heap (please refer to Figure 8 frequently while reading the remainder of this chapter.)

- Communication Manager: The Communication Manager (labeled cvrnCommunicationManager_c after the C++ class used to implement it) maintains the networking connections that are created and removed by Nexus.
- Active Key Manager: The Active Key manager (activeKeyManager_c) manages the creation and removal of keys; the loading of possibly persistent keys into memory; and contains information on which remote IRB links subscribe to the data.
- Persistent Heap: The Persistent Heap dynamically allocates memory for the active key manager and performs reads and writes to persistent store. It basically provides the database capabilities of the IRB.

In addition to these three main components are three supporting components: the Incoming Hailing Channel, the Channel Buffer, and the Garbage Collector.

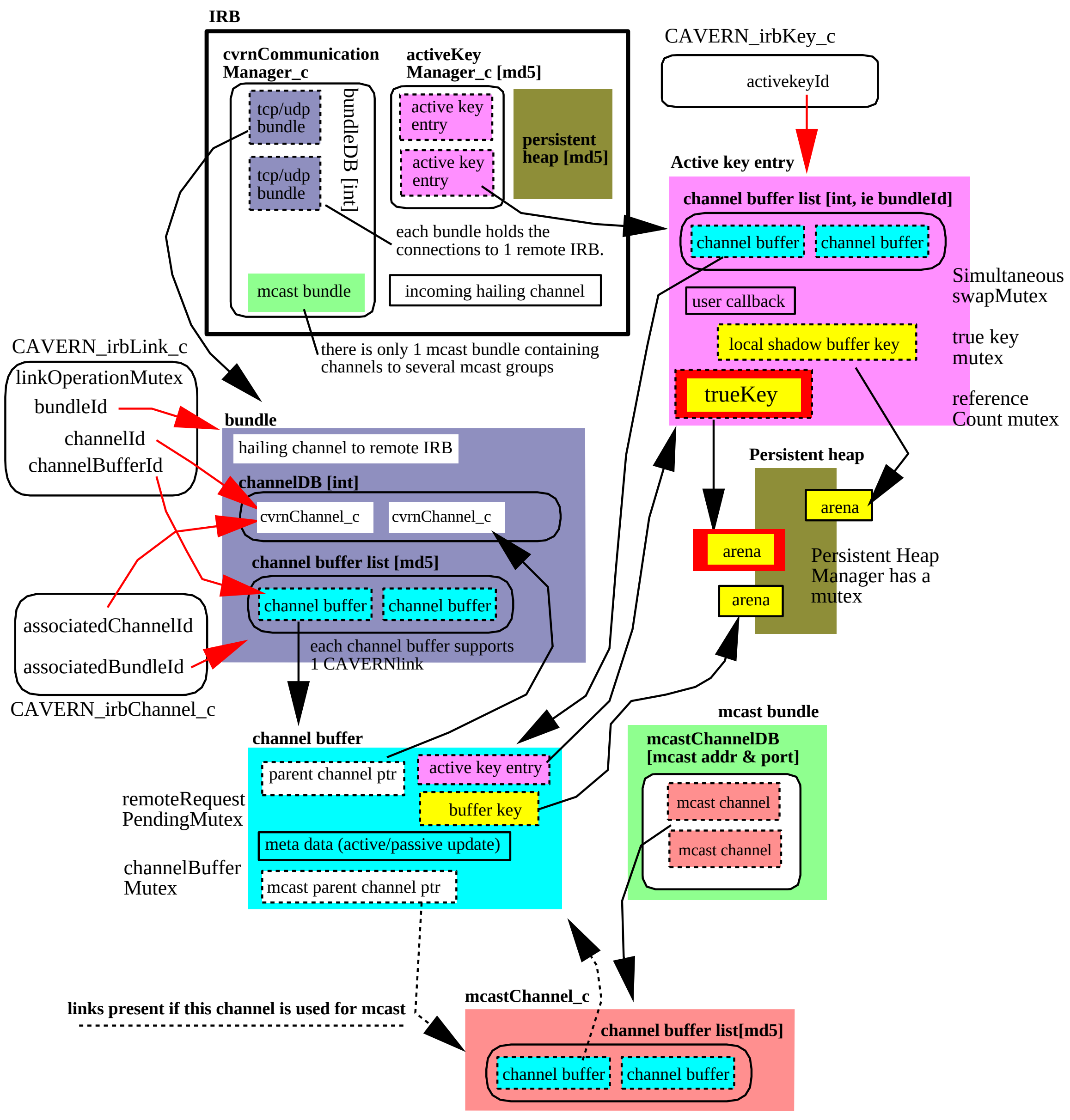


Figure 8. Internal Structure of an Information Resource Broker

- Incoming Hailing Channel: The incoming hailing channel is always opened when the IRB is first initiated. This channel listens on a pre-specified port (10000) for incoming hails from other remote IRBs. This is implemented by opening a Nexus Endpoint to which other IRBs may attach. These hails are command messages to coordinate the remote IRBs on a variety of future tasks that fulfill the CAVERNsoft API. That is, they coordinate the creation and removal of communications channels of varying networking protocols and they coordinate the creation and removal of remote resources needed to sustain links between local and remote keys.

- Channel Buffers: Whereas the active key manager accesses the persistent heap directly through its API, the communication manager communicates with the active key manager through a series of channel buffers. These channel buffers are created whenever a link of two keys are requested by the user. They are the primary means by which active keys can determine which networking interface must be used to send outgoing data destined for a remote key. They are also the primary means by which incoming data determines which active key should receive the data.

- Garbage Collector: When connections and, hence links, are broken they are not immediately removed from the IRB. Instead they are marked by the system as **defunct**. The garbage collector is a concurrent thread launched by the IRB to monitor itself from time to time (every 10 seconds) to search for defunct resources and remove them.

  The IRB does not remove a resource the moment it is labeled as defunct, because there are many possible concurrent threads also attempting to access the same resource. Removing the resource while it is being used will very likely crash the IRB. Instead the concurrent threads are allowed to also determine for themselves that the resource is defunct and cleanly abort their current operation. Marking the resource defunct prevents further access from any new threads of control.

As part of the garbage collector's normal operation, it will send out periodic "health pulses," which consist of small packets of data, to other connected IRBs. This is necessary because Nexus can only determine that a connection is broken by sending data over it.

4.2 Communication Manager

The communication manager consists of a dictionary (hash table) of communication bundles to support point-to-point connections, and a single multicast bundle to handle multicast connections.

When two IRBs communicate with one another through a point-to-point connection, they must each hold a communication bundle (Figure 9.) This bundle will maintain all the networking connections that will be created between the IRBs. For example a bundle will hold one connection via a reliable protocol, and another connection via an unreliable protocol. Each communication bundle contains a Nexus Startpoint that is attached to the remote IRB's Incoming Hailing Channel Endpoint. Using this Startpoint the local IRB may send command messages to the remote IRB.

Each of the separate networking connections that are created between two IRBs are encapsulated in a communications channel (labeled cvrnChannel_c). This channel contains Nexus

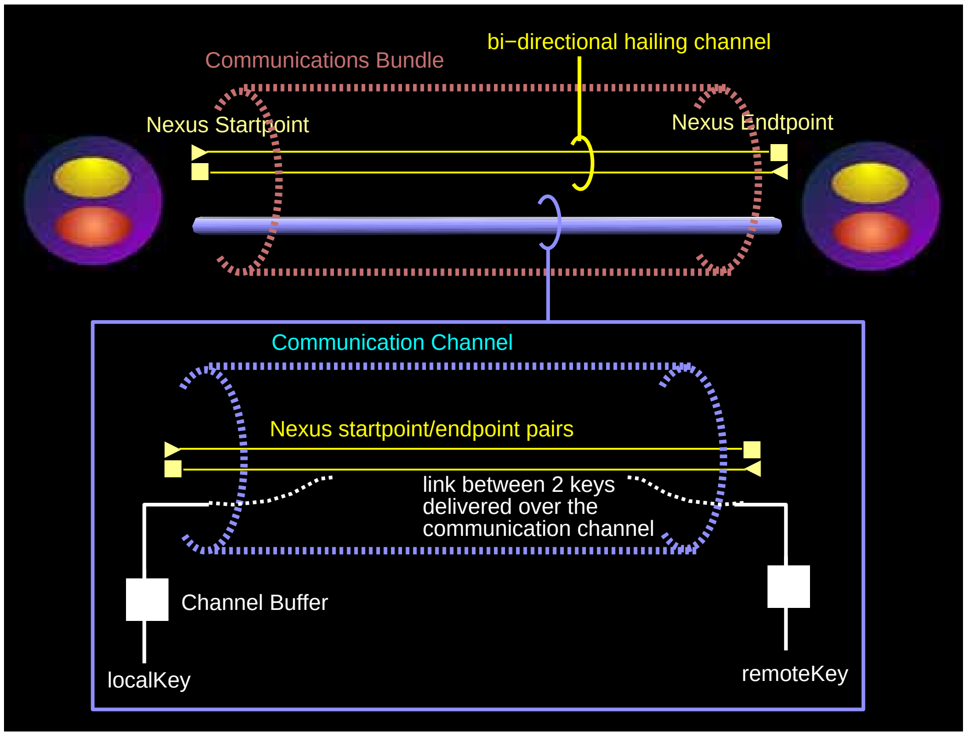

bi−directional hailing channel
Communications Bundle
Nexus Startpoint
Nexus Endtpoint
Communication Channel
Nexus startpoint/endpoint pairs
link between 2 keys delivered over the communication channel
Channel Buffer
localKey
remoteKey

Figure 9. A Communications Bundle consists of a hailing channel and multiple Communications Channels. Each channel manages a number of links that bind local and remote keys together.

Startpoint and Endpoint pairs that connect to a corresponding communications channel and bundle on a remote IRB. These communication channels are maintained in a channel dictionary (hash table labeled channelDB).

In addition to the channel dictionary, the bundle also has a single channel buffer list (again a hash table) which holds pointers to the channel buffers. The channel buffer is used to link the communications channel to the key containing the data (more on this later). One channel buffer is created per CAVERN link created. Each channel buffer in the bundle is uniquely identified by the key it is associated, and hence a key cannot be redundantly linked to the remote IRB. A key may however be linked more than once to completely separate IRBs since each of these links (and hence channel buffers) will reside in separate communications bundles, each with their own channel buffer lists.

4.3 Active Key Manager

The active key manager is a dictionary of all the keys that are currently active in the IRB. Whenever the user defines a key locally, or whenever a remote IRB requests a link to a local key, an entry is made in the active key manager. A reference count is maintained in the entry so that the IRB will know when the key is no longer being used and hence may be purged to free valuable memory. Each active key entry consists mainly of a channel buffer list (much like the one in the bundle), a trueKey pointer and a local shadow buffer key pointer. Auxiliary items include a pointer to a user-defined callback which can be triggered whenever new key data arrives; a number of mutual exclusion variables to guarantee atomicity in data transfer.

The channel buffer list is a dictionary of pointers pointing to the same channel buffers in the bundle's channel buffer list. As mentioned earlier the channel buffers are the interface between the communication manager and the active key manager.

When the user requests that a key be linked to a remote key over a specific communications channel the IRB first identifies the bundle that contains the channel and creates a channel buffer. An entry in the bundle's channel buffer list is created to point to this channel buffer. In the bundle's channel buffer list the entry is uniquely identified by the name of the key. The active key manager locates the active key that is being linked and adds a similar pointer to the channel buffer, to its channel buffer list. In this case the channel buffer entry is uniquely identified by the communications bundle's ID rather than the name of the key.

As mentioned earlier the active key entry also contains a trueKey pointer and a local shadow buffer key pointer. These pointers point to arenas of memory allocated by the persistent heap. The trueKey's arena contains the actual current data stored in a user-defined key. The local shadow buffer key's arena is used as a cache for any locally initiated user requests to fill the key with new data. When the user initiates a put() call to the key, the shadow buffer is first filled with the new data. When the data has been deposited, the trueKey is locked and a pointer-switch occurs to swap the contents of the trueKey and the shadow buffer key. The trueKey, now holding the new data, can be unlocked for general access. The shadow buffer on the otherhand now holds an arena that can be re-used for the next put()- **hence the same memory is efficiently recycled reducing the need for dynamic memory reallocation**.

The channel buffer also contains a similar buffer key with its corresponding pointer to an arena of memory in the persistent heap. Any incoming data from the network is first collected into the channel buffer's key buffer. Then, as in the case with the local put() call, a pointer-switch occurs to swap the contents of the channel buffer's buffer key and the trueKey. This scheme allows the IRB to minimize the number of redundant memory copies that are needed to move data from the network to the user and vice versa. In addition it also **guarantees that all data accessed by the user and transmitted to remote IRBs are atomic**. Finally, **since one channel buffer is allocated for each remote subscriber to the key, the IRB is able to parallelize the download of incoming data streams.**

4.4 Putting it all together

4.4.1 When a user places new data in a key

When a user invokes the put() call to place new data into a key the key is first located in the active key manager. The data is deposited in the local shadow buffer and then pointer-switched with the trueKey. Finally the IRB iterates through the list of channel buffers in the active key's channel buffer list and sends the data out to any IRBs that may be linked (subscribing) to the key. This is done by examining each channel buffer, and following the channel buffer's parent channel pointer to the channel responsible for delivering the data to the external IRB. Data transfer can then occur by accessing the Nexus startpoint stored in the channel. This process currently occurs sequentially. In principle this process can be done concurrently via multiple threads.

4.4.2 When data arrives from a channel

When new key-data arrives from a channel the message usually contains the name of the key and the ID of the bundle that holds the communication channel. With this information the bundle can be retrieved from the bundle database. With the bundle at hand, the key can be looked up in the channel buffer list. If found the channel buffer is retrieved and the data is deposited in the channel buffer's buffer key. Recall that by depositing the data in the buffer key, the IRB is able to parallelize the receipt of incoming data destined for the same key, as each channel has its own channel buffer.

When the data has been completely transferred, a pointer-switch occurs to swap the data in this buffer with that in the active key entry's trueKey. If the user has specified a callback in the active key it will fire so that the client application can be alerted. Then the IRB must scan the active key's channel buffer list and iterate through each of the channel buffers and send the data out to all their associated channels. Hence **local user notification is prioritized over data retransmission to the external subscribers**.

4.5 Keytool: the Persistent Heap

The Persistent Heap is implemented as a prototype to determine the kind of API needed to support the IRB. As a result the implementation is not optimal. In the future the persistent heap will be implemented with more efficient and robust database technology.

The heap identifies the individual storage arenas by a path name and a final leaf (key) name, that corresponds directly with a UNIX local path and a filename. Hence when a key is made persistent, a sequence of UNIX sub-directories (corresponding to the path specification of the key,) is created before the final data is stored at a filename with the same name as the key. In addition a meta-data file is created that is used to store any auxiliary data about the key (such as time stamps).

When a key is first requested the persistent heap searches the file system for the same pre-existing key. If it exists the persistent heap will load it into main memory as an arena. This arena is turned over to the active key manager to be used as needed. When new data is deposited in the arena the persistent heap first determines if the key will fit in the arena already allocated to the key. If it does, the **logical** size of the arena is set to the size of the key- the **physical** size of the arena is unchanged. If it does not fit (i.e. it is too big) then the arena must be grown to fit the key and the old arena is discarded. **This resizing scheme is done to reduce the need for constant dynamic memory reallocation by the operating system** (which can be very slow). Instead the arena will always "stretch" to the size of the largest piece of data encountered by the key. Currently the only provision for re-shrinking the arena size is if the IRB is re-started or the key is undefined and then redefined. On re-start the IRB will adopt the size of the last committed version of the key.

Keytool allows independent keys to be read, updated and committed in parallel. Keytool commits keyed data by first writing the data to a temporary file and then renaming the file to replace the previously existing one. It does this to provide a small measure of fault tolerance in the system so that if a commit is interrupted during a file write there is always a backup version available.

As a side note about keys: for efficiency, all path names and key names are concatenated together and condensed into a 16 byte MD5 key (MD5 is an encryption algorithm that allows arbitrarily sized bodies of data to be encrypted into a unique 16 byte value). All keys used in the IRB are MD5 encoded. All dictionaries that use a key's name as a search key use the MD5 encoding as the key's ID (Figure 8 has labeled all dictionaries that use the MD5 encryption scheme with [md5].) This encoding scheme is not used for encryption per-se, rather it is used as a means to uniquely represent a key name in a statically sized addressing space.

4.6 Multicasting

Multicasting involves clients all sending data to a single multicast group address. Any clients that happen to be listening to the same address will automatically receive a copy of the data. This is useful because it reduces the amount of data each client must broadcast to all its linked clients.

This definition of multicasting however is somewhat incompatible with the key-link scheme used so far to distribute data between IRBs. Thus far the IRB has supported point-to-point broadcasting. This has been achieved by allowing keys to be linked over communications channels between a number of distinct IRBs. This has allowed applications to link a key with one name to a remote key with an entirely different name. The consequence of this is that key names are automatically globally unique, as they are identified by the path, the name and the IRB in which they reside. This ability to link multiple keys with differing key names is essentially a form of name aliasing. When applying this scheme to multicasting, name aliasing is inefficient as it ultimately means the multicast channel is being used for point-to-point data delivery. Hence in the IRB, when a multicast channel is used, no name aliasing is allowed. Hence applications that wish to deliver data over multicast must police their own global naming scheme so that key names that mean one thing on one IRB mean the same on all the participating IRBs. Hence when data is sent out on multicast the name of the key can be attached to the message and properly distributed to all the participating IRBs.

To provide for this each IRB contains a single multicast bundle that maintains all the multicast channels to which the IRB subscribes (Figure 10.) Each channel contains a channel buffer list whose entries are uniquely identified by the name of the key (encoded as an MD5 key). And as with the point-to-point scheme, each channel buffer will subsequently point to their associated active key entries.

When data arrives on a multicast channel, the destination key of the channel is extracted and used to locate the respective channel buffer. Once found, the data is deposited in the channel buffer's key buffer. When done, the key buffer is swapped with the contents of the trueKey buffer in the active key entry. This will then initiate a sequence of retransmissions that may be necessary due to subscriptions by other IRBs to the key (as in the point-to-point case).

4.7 Providing Fault Tolerance

One design goal of the IRB was for it to be able to remain persistent for great lengths of time. Hence it makes every effort to not crash or terminate a program when channels and links

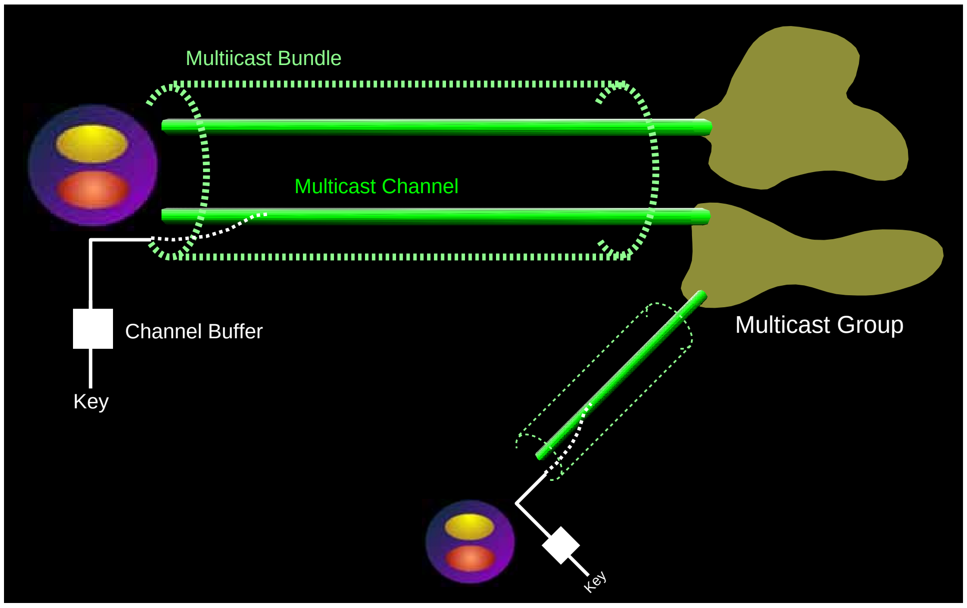


Figure 10. A Multicast Bundle manages multiple multicast channels that subscribe to independent multicast groups.

are closed or broken. Instead it offers callbacks to user-specified functions that can be used to alert the program of such events. The program can then proceed to attempt to re-establish the channel. At the current time this must be done manually by the user. In the future, the IRB will attempt to re-establish connections automatically.

Hence currently when the client application receives an event that a channel has been broken, the protocol the client should follow is to delete and create a new channel object; then delete all the associated link objects and finally recreate all the new links with a newly opened channel (either to the same server or perhaps to a backup server.) The keys need not be redefined. This allows the client application to continue functioning even in the event of a crash on a remote server. The IRB however cannot protect the client program against any bugs that may be inherent in the program- such as accidental accesses to dangling memory pointers. These will surely crash any program (regardless of whether it is IRB-based.) This however will not adversely affect the remote IRB to which the client may have been connected. The crash will eventually be noticed by the IRB and the previous resources occupied by the connection, recycled.

As part of the IRB's fault tolerance provisions, all IRB interface API calls return status values that state whether a link, channel, or key is no longer valid (i.e. stale.) These statuses are provided to prevent the client programs from accidentally accessing dangling pointers that may have once referred to valid objects. This allows the IRB to dynamically purge broken connections and reclaim system resources without adversely affecting the client program. At the outer perimeter of Figure 8 one can see three of the IRB API's protective classes (CAVERN_irb_Link_c, CAVERN_irbChannel_c, CAVERN_irbKey_c.) These classes are the client program's handles to the internal structures of the IRB. A detailed description of the IRB API is provided in Appendix A.

4.8 Compatibility between the IRB and the CAVE library

The IRB does not depend on any part of the CAVE library. It was designed as a system that can be attached to graphical as well as non-graphical programs. The official CAVE library and its variations use similar models for separating computation and graphics rendering. That is, when a CAVE application is launched the CAVE library will first fork-off a number rendering processes (one for each wall of the CAVE,) and a tracker process. Each of these processes, including the main process, communicate with each other using shared memory. This model is somewhat different from the threaded model of the IRB. The IRB, when launched will thread-off a number of concurrent threads. Threads are similar to separate forked processes in that they allow multiple tasks to operate concurrently. Threads are different from forked processes in that they take less processing time and resources to create. In addition they share the same address space as the main program, hence there is no need to use shared memory for interprocess communication.

However in order for the CAVE forked model and the CAVERNsoft threaded model to work together the IRB can only be initiated after the CAVE library has forked-off its many processes (after the CAVEInit() call). In addition the IRB can only be used in the CAVE's main processing loop as it has no scope in the separate rendering processes. This is acceptable because the rendering processes should be dedicated solely to rendering tasks anyway.

As the IRB is a threaded library (it uses Posix Threads for its implementation) the mechanisms for mutual exclusion are incompatible with those used by the CAVE library. The CAVE library uses shared memory locks that are part of Silicon Graphics' shared memory library. The IRB library uses the Pthreads mutual exclusion mechanism. The two are not compatible. The CAVE library provides a lock management API to allow applications to create read and write locks that are shared across processes (including the rendering processes). The IRB provides a lock management API that is a simplification of the Posix standard that allows locking of threads within the main process. Hence, as long as the application uses the IRB's locking mechanism to lock data within the scope of the CAVE main process, but uses the CAVE's locking mechanism to lock data between the main process and the rendering processes, there should be no conflict.

Ultimately a solution would be to move the CAVE library to a threaded model. Since operating systems such as Windows-NT use a threaded model, creating a threaded CAVE library would also simplify the port across multiple platforms- especially when a uniform threaded API such as ACE(33) is used. However at this time the main computing platforms for the CAVE are Silicon Graphics computers which is only beginning to provide stable Pthread implementations. The current versions offer little control for scheduling threads across multiple processors. This capability is crucial for the CAVE because the CAVE library tries to dedicate independent processors to each forked process.

CHAPTER 5

EVALUATION

The IRB is neither strictly a networking library, nor a distributed database, nor a distributed shared memory. It unifies networking and databases in an unconventional way to provide persistent distributed shared memory services over both reliable and unreliable networks. The claim is that these properties make it particularly well suited to distributing a wide spectrum of teleimmersive data. To answer to this claim, this chapter will first begin by illustrating, through example, the ease with which the IRB provides networking and database capabilities to its users. This includes a brief report on the integration of the IRB into two CAVE applications, NICE and General Motor's VisualEyes. Following this, a critique of the IRB's current weaknesses will be presented along with a comparison of the IRB to HLA (the Department of Defense's competing product.) Finally the results of a set of networking benchmarks comparing the capabilities of a typical TCP program, Nexus and the IRB, are presented.

5.1 Benefits of using IRBs for Teleimmersion

5.1.1 IRB Network and Database Programming is Easy

As the IRB combines both networking and database capabilities into a coherent interface, programming with it is relatively easy. The API allows programmers to treat data sharing in a way that they are most familiar. That is, they may treat the IRB as a persistent distributed shared memory, or they may use the trigger mechanisms and treat the IRB as a message passing system.

As an example of the ease of IRB programming the following is a complete IRB-based program. Once this program is compiled and executed it will automatically possess the ability to serve remote connections from other clients and link and share information between keys. Furthermore if no remote clients communicate with this program it will occupy very few system resources.

```
#include "CAVERN.h++"
#include <unistd.h>

main(int argc, char** argv)
{

/// Startup CAVERN.
CAVERN_irb_c *personalIRB = CAVERNInit(&argc, &argv, NULL);
if (!personalIRB) exit(1);

/// Now you can do your own stuff here. Or you can do nothing in a
/// an infinite while loop.
while(1) sleep(10);
}
```

For a more complex example, the following are the main fragments of a program to download data stored in a key (perhaps a 3D model) on any remote IRB or IRB-based application regardless of what the application was originally designed to do. Figure 11 illustrates each of the steps.

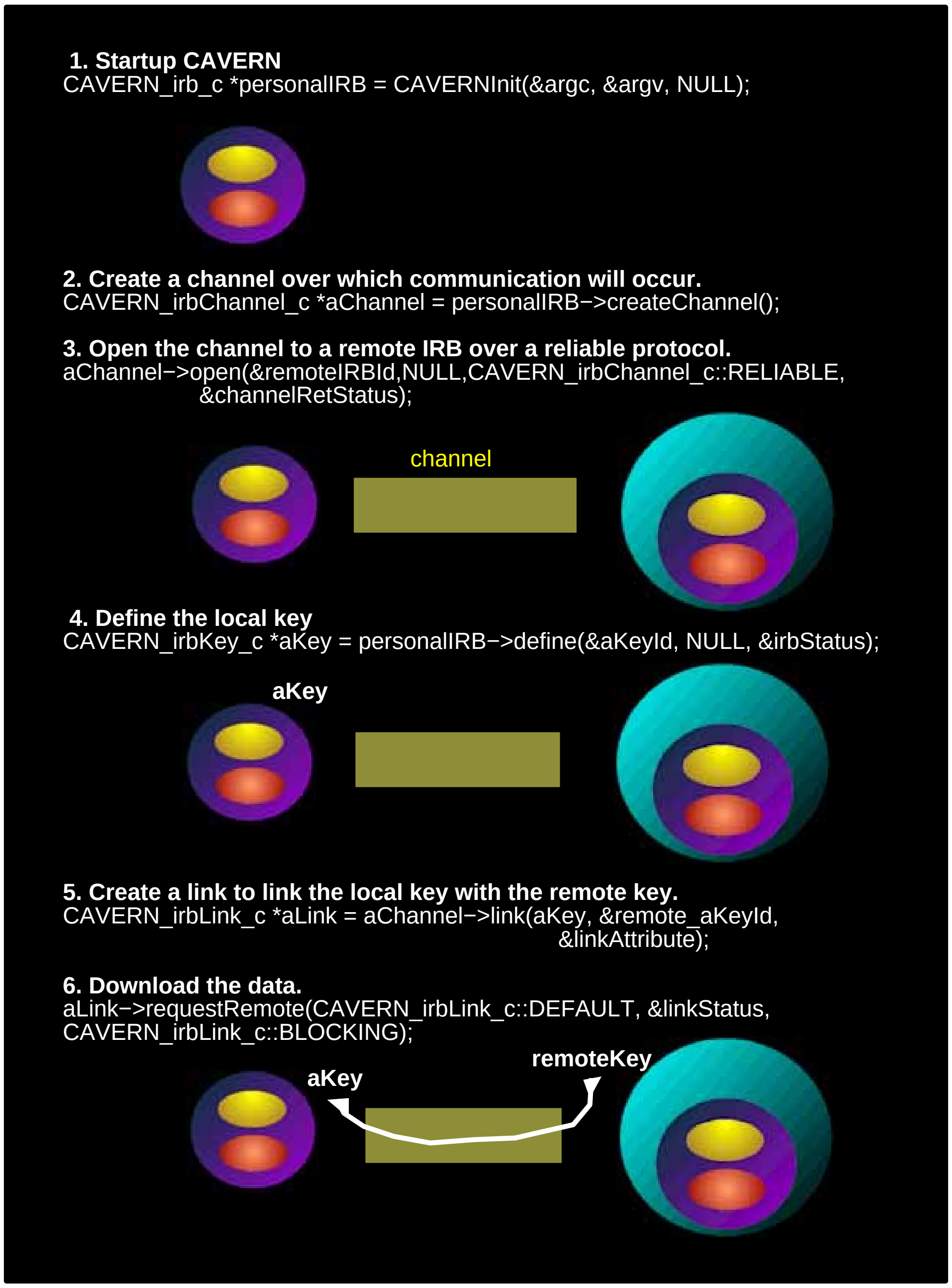
```
/// 1. Startup CAVERN
CAVERN_irb_c *personalIRB = CAVERNInit(&argc, &argv, NULL);
```

```
/// 2. Create a channel over which communication will occur.
CAVERN_irbChannel_c *aChannel = personalIRB->createChannel();

/// 3. Open the channel to a remote IRB over a reliable protocol.
aChannel->open(&remoteIRBId,NULL,CAVERN_irbChannel_c::RELIABLE,
&channelRetStatus);

/// 4. Define the local key
CAVERN_irbKey_c *aKey = personalIRB->define(&aKeyId, NULL, &irbStatus);

/// 5. Create a link to link the local key with the remote key.
CAVERN_irbLink_c *aLink = aChannel->link(aKey, &remote_aKeyId,
&linkAttribute);

/// 6. Download the data.
aLink->requestRemote(CAVERN_irbLink_c::DEFAULT, &linkStatus,
CAVERN_irbLink_c::BLOCKING);
```

IRB-based applications treat each other as information resources that they may access at any time. Hence IRB-based programs take on almost biological abstractions where IRBs may dock with other IRBs with matching key names much like enzymes are able to target specific molecular structures.

5.1.2 Topology Building is Easy

Since the IRB handles many remote connections transparently it can be used to easily build a variety of networking topologies. For example lets say a set of teleimmersion applications chose to set a single key called AVA as the key to distribute avatar tracking data for everyone. Figure 12a. shows an IRB server connected by a client program that links the client's local AVA key with server's AVA key. Any time the client places new data in local key AVA, the



1. Startup CAVERN

```
CAVERN_irb_c *personalIRB = CAVERNInit(&argc, &argv, NULL);
```

2. Create a channel over which communication will occur.

```
CAVERN_irbChannel_c *aChannel = personalIRB->createChannel();
```

3. Open the channel to a remote IRB over a reliable protocol.

```
aChannel->open(&remoteIRBId,NULL,CAVERN_irbChannel_c::RELIABLE,
                &channelRetStatus);
```

4. Define the local key

```
CAVERN_irbKey_c *aKey = personalIRB->define(&aKeyId, NULL, &irbStatus);
```

5. Create a link to link the local key with the remote key.

```
CAVERN_irbLink_c *aLink = aChannel->link(aKey, &remote_aKeyId,
                                          &linkAttribute);
```

6. Download the data.

```
aLink->requestRemote(CAVERN_irbLink_c::DEFAULT, &linkStatus,
CAVERN_irbLink_c::BLOCKING);
```

Figure 11. Linking two keys over a channel between two IRB-based programs.

same change will be propagated to the server. This is done with code fragments such as the following:

```
/// Startup CAVERN.
CAVERN_irb_c *personalIRB = CAVERNInit(&argc, &argv, NULL);

/// Create a channel over which communication will occur.
CAVERN_irbChannel_c *aChannel = personalIRB->createChannel();

/// Open an unreliable channel to a remote IRB.
aChannel->open(&remoteIRBId,NULL,CAVERN_irbChannel_c::UNRELIABLE,
&channelRetStatus);

/// Define the local Avatar key
remote_aKeyId.setName("AVA");
CAVERN_irbKey_c *aKey = personalIRB->define(&aKeyId, NULL, &irbStatus);

/// Create a link to link the local key with the remote key.
CAVERN_irbLink_c *aLink = aChannel->link(aKey, &remote_aKeyId,
&linkAttribute);

/// Allow new incoming data to call a user-defined callback.
aKey->trigger(callbackForAKey,NULL);
```

Now by using this exact same program another copy of the client (client B) may attach to the IRB server. Any changes to the AVA key on client B will be echoed to the server which will relay the change to client A (Figure 12b.)

Furthermore client A and client B may be located at very distantly located sites each with small clusters of computers that may also wish to join in the teleimmersive session. They may all either connect directly to the server, or they can connect to their nearest client (Figure 12c.) All this is achieved by the same unmodified program. Yet another possibility is for both client

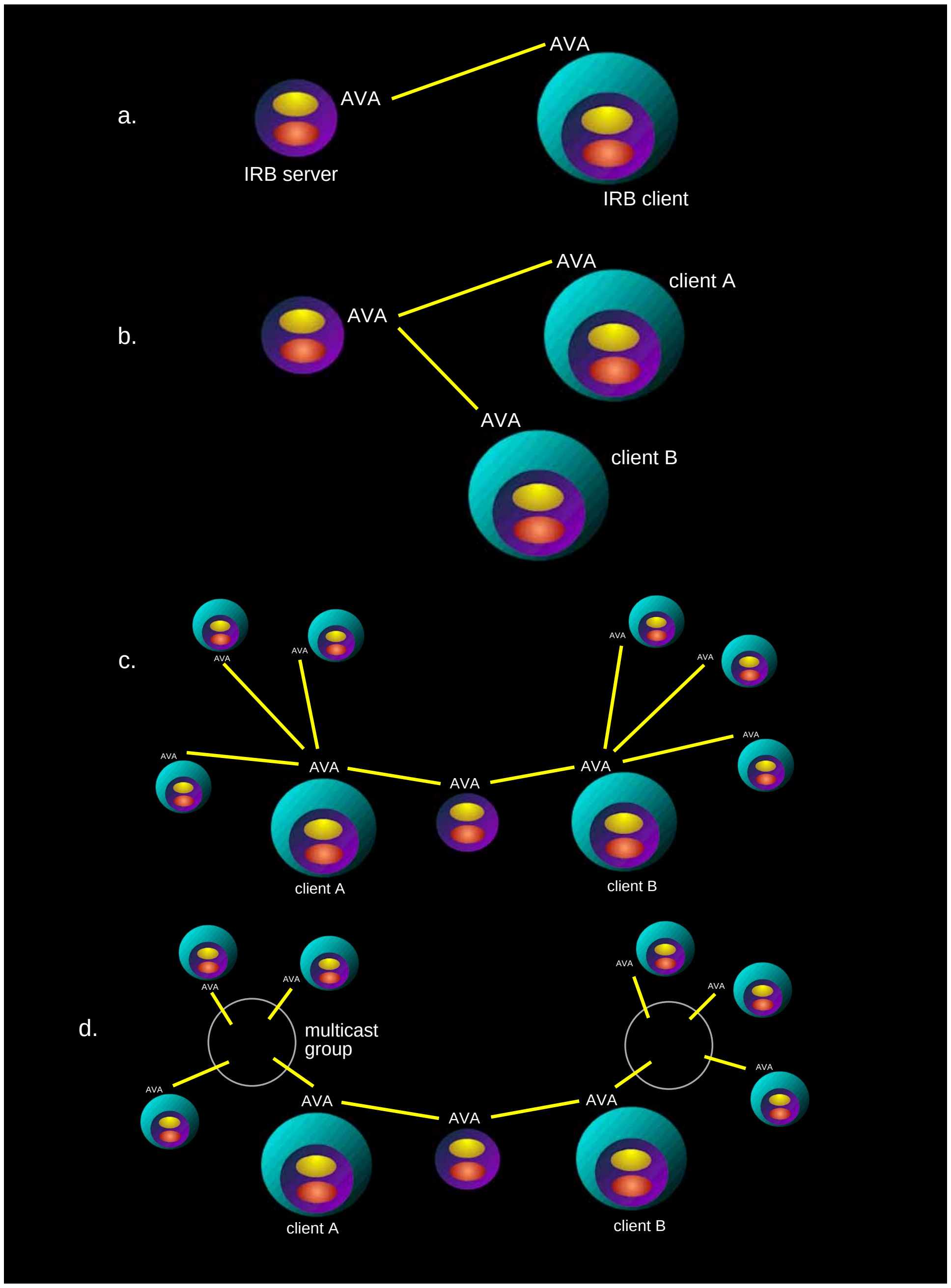


Figure 12. A variety of schemes for sharing avatar information in CAVERNsoft

A and client B to link their AVA keys to multicast channels so that their local area clients may simply tune into the AVA keys via the channel (Figure 12d.) Hence any changes to AVA on any of the client programs will be propagated to, say, client A which will then be propagated towards the central server which will further propagate the change to client B and finally client B over multicast to its subscribing clients. In fact the central server can be removed completely and client A and client B may connect directly- hence instantly producing a multicast tunnel.

5.1.3 Persistence Makes it Easy to Save Downloaded 3D Models

When a remote key containing a 3D model (or any piece of data) has been downloaded (as in the second example in section 5.1.1) it is particularly easy to commit the model to persistent store. The only command that needs to be added to the example is:

```
aKey->commit();
```

When the client application starts up again at a later time and re-defines the key, the key will automatically be filled with the committed data. To check for new remote updates the client need only link the key to the remote IRB. The IRBs will compare timestamps and appropriately synchronize versions.

5.1.4 Concurrency Affects Graphics Minimally

A number of casual tests uploading 8 Megabyte keys to a remote IRB-based Performer CAVE application showed that it had very little impact on the graphics during the upload of the data to the IRB and during the commit of the data to persistent store. The IRB performs these operations in concurrent threads which will take advantage of spare processors if available.

5.1.5 Fault Tolerance Allows Applications to Survive Server Crashes and Re-route to Backup Servers

Since the transfer of information is via storing new data values in the keys the break of a connection due to a possible server crash will not cause the client application to crash also. This is because the handles for data distribution are keys and not network connections. Connections may break without affecting the keys. The application can continue to operate by placing data at- and retrieving data from the keys even though those changes may not be propagated to the remote site.

When the client application re-establishes the connection, to possibly a backup server, and then relinks the local and remote keys, normal operation of the program can resume.

5.2 Applications of the IRB

The following sections will describe two slightly differing approaches to using CAVERNsoft to support Teleimmersion. The first (retro-fitting NICE) demonstrates the use of CAVERNsoft as a message passing library. The second (retro-fitting General Motors' VisualEyes) demonstrates the use of CAVERNsoft as a distributed shared memory and database.

5.2.1 NICE

The original NICE was a highly collaborative application that used reliable TCP, unreliable UDP and multicast for transmission of world state information, avatar information, and three dimensional models. Each of these employed an independent mechanism for communications and hence was coded inconsistently. Retro-fitting NICE with CAVERNsoft provided a means to unify these communications mechanisms. In the new version a central IRB server was used both to support the garden's simulation as well as to broadcast avatar information.

To support avatar management a low-level CAVERNsoft avatar class was written to replace the original NICE avatar class. This class subscribed to two keys. The first key, the Avatar Hailing Key (AHK) was linked to an IRB server via a reliable connection. It is used as a message passing conduit through which information about newly joining avatars and exiting avatars are distributed. Hence any time a new client entered the NICE world, it would send a message on this conduit announcing its presence to others. The second key, the Avatar Tracker Key (ATK) was linked to an IRB server via an unreliable connection. This key allowed the distribution of avatar state information- mostly from the CAVE/Idesk tracker.

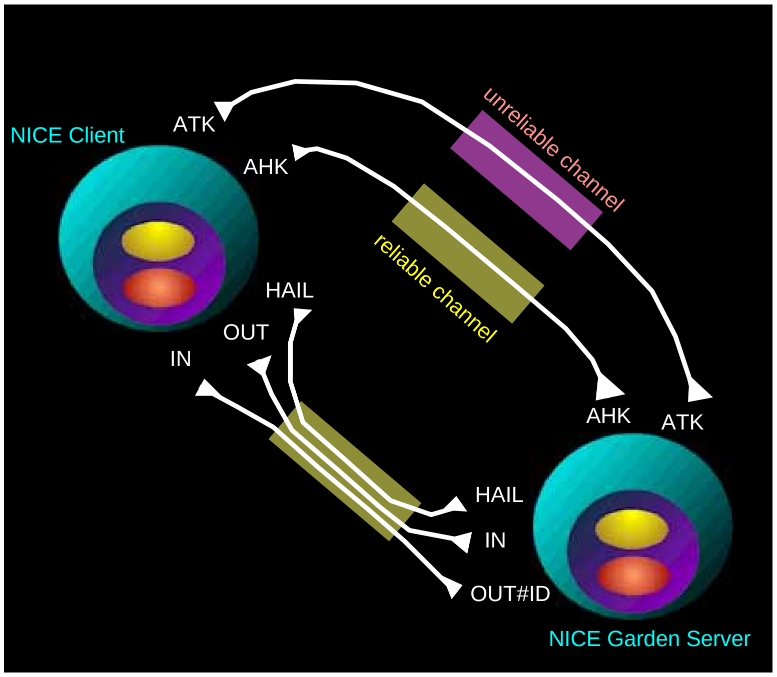


Figure 13. NICE Client Connecting to a Garden Server. OUT#ID is a dynamic key generated by the remote NICE client to allow the garden server to send it private messages.

To support the relaying of garden state information, each client created three keys, a hailing key, an incoming data key, and an outgoing data key. The hailing key was used to send initial identification information to the IRB garden server. In response the garden server would subscribe to a separate outgoing key (OUT#ID) that was dedicated to delivering data only to the connected client via the client's incoming data key. Each client would send data via its outgoing data key to the IRB server to be received by its incoming data key. These two keys provided the main message passing channels for relaying the state of the garden.

The original NICE architecture was programmed serially. That is, all networking was performed at specific instances in the NICE main loop. This meant that NICE had few provisions for guarding against parallel accesses to its internal data structures. CAVERNsoft on the otherhand was coded entirely with concurrent programming in mind; this made the retro-fitting process difficult. Two solutions were considered: the first solution would involve painstaking examination of each line of NICE source code to analyze situations where CAVERNsoft might create a race condition; the second solution would involve providing CAVERNsoft with an ability to better support serial applications while leaving much of its operations parallel. The latter solution was chosen as it is anticipated that the same situation will arise when attempting to retro-fit other CAVE applications.

Providing serializing support for NICE resulted in the addition of a CAVERNsoft C++ class called CAVERNplus_callSequencer_c. This class when invoked allowed CAVERNsoft's event triggers to fire only at specific instances. These instances would be controlled by the NICE main loop. Hence, while CAVERNsoft was still able to process incoming networking data in parallel, the notification of the arrival of new data was performed at controlled locations in the NICE main loop. This in effect emulates the serial functionality of the original NICE but with one major difference. In the original version of NICE, the serial processing of networking messages mean that NICE could freeze in the event of a network stall. This occurs when NICE is in the middle of transmitting or receiving a packet of data over an unreliable network. This problem is eliminated in the CAVERNsoft implementation because all networking is processed concurrently.

5.2.2 General Motors - Teleimmersive VisualEyes

A routine part of the car design process at General Motors includes the time consuming task of building large clay models of their designs. These clay models reside in the design studios over long periods of time as the prototype that they will repeatedly refer to while they are considering new design changes. In recent years GM has been interested in producing the electronic substitute for the persistent and evolving clay model, in the hope that it will take them less time to modify the electronic representation than the clay model, and that the final version will be both precise in terms of engineering specifications as well as aesthetic requirements.

To transition to the electronic clay model, GM has developed VisualEyes- an application that allows GM designers to import 3D CAD models into the CAVE for quick visual inspection and design reviews. This initial use of CAVE-based technology has generated considerable interest in other GM research sites around the world, all of whom are planning their own CAVE installations.

This has prompted GM to collaborate with EVL again, to extend VisualEyes to allow GM's trans-globally situated research and design teams to collaborate in remote design reviews. The goal is to allow designers to both synchronously and asynchronously access a design that persists and evolves over time.

When working with trans-globally situated collaborators where time-zone differences may significantly limit the opportunity for synchronous collaboration, the need to provide asynchronous support increases. Collaborators will need to be able to record versions of designs,

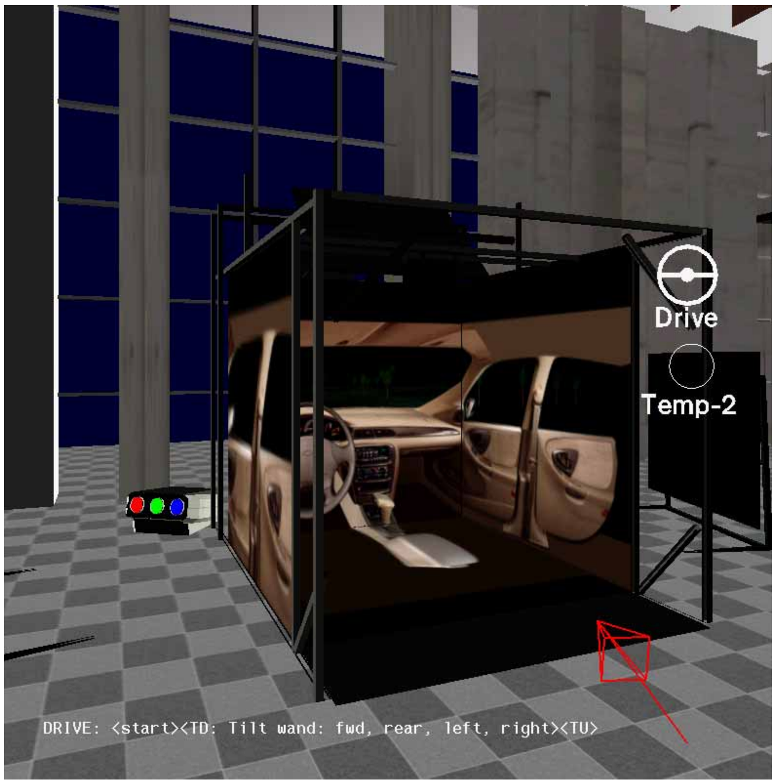


Figure 14. General Motors- VisualEyes. Selecting the CAVE allows the participant to teleport into the environment where the interior of a GM car is being designed.

and annotate parts of designs for later review. In essence what is required is a persistent teleimmersive environment which has the ability to chronicle design evolutions.

The retro-fitting of VisualEyes (VE) for collaboration began with the use of the same CAVERNsoft avatar template used for NICE for managing the entry and exit of avatars. Whereas NICE had the ability to dynamically load objects into its scene graph, VisualEyes did not. This meant that all avatars in VE had to be pre-loaded and made invisible until they are invoked. Hence when a new participant entered the space the participant's ID would trigger the appearance of the respective avatar.

As part of a teleimmersive demonstration at the Supercomputing '97 conference, GM wanted to illustrate the idea that one could use a networked CAVE as a portal to the design spaces at other CAVE's/Immersadesks, at other remote GM design centers. Hence the prototype allowed participants to aim their wand at a virtual CAVE or Immersadesk depicting a remote site, and press a button to enter the space. In VE this was implemented by a switch in its scene graph when the button was pressed. This in turn altered the avatar information that the participant was sending so that the connected CAVEs and Immersadesks knew whether the participant had left or entered the space. Hence participants were able to jump in and out of various design spaces much like in a VRML browser.

Finally to share updates of each world, the script files that VE loaded on startup were marked with labels that registered the objects that were collaboratively manipulable. These labels were conveniently mapped to keys of the same name in CAVERNsoft. Hence when a participant moved an object the new value of the change would be stored in the respective key and all participants subscribing to the key would receive the update which would in turn fire an event to adjust their local scene accordingly. Since these keys were mainly stored in a central IRB-based server any participant may enter the world and receive an update of the world's current state from the server. In addition a remote commit can be performed on each key so that the next time each participant entered the space the VE clients will synchronize their state according to the central server's committed values; hence the environment is persistent supporting both synchronous and asynchronous collaboration. Based on this initial experiment GM has proposed that the IRB's underlying database hierarchy be a canonical data structure to allow VE to dynamically interoperate with Alias (a CAD modeling and animation package.)

As a side note: because of the IRB's symmetric client/server properties the central server used in the GM application need not be a special server written for the application. Any IRB-based program (even the NICE garden server) will do.

5.3 Areas of Potential Improvement in the Prototype IRB

The IRB bases all of its networking capabilities on Nexus. As a result the IRB is able to leverage the capabilities of Nexus as they are implemented, however, at the same time it suffers from some of the current limitations of Nexus.

5.3.1 UDP and Multicasting

Currently Nexus (version 4.1) only supports the transmission of UDP and multicast packets of 1024 bytes however the Nexus group is currently working on a scheme to allow Nexus to transmit packets of arbitrary size. Nexus will handle large packet disassembly and re-assembly.

5.3.2 New Threading Model

Nexus currently provides very little control of threads. Threads are used in the IRB to handle multiple parallel streams of incoming data. Nexus' current threading model allows either completely non-threaded, or fully threaded communication. In non-threaded communication all communications channels are treated completely serially and hence no parallelism is possible. In fully threaded communication each incoming remote service request is threaded. This poses a problem for the IRB because if the IRB is flooded with many rapidly arriving UDP packets, the resulting spawned threads will overflow the limit allowed by the operating system.

The ideal threading model for the IRB is for each communications channel that is opened by the IRB to be managed by a single thread. This single thread can then create additional threads to allow parallel processing of data streams. However these threads should be managed so that they come from a limited pool of threads. The number of threads in the pool can grow and shrink based on the application's specification or based on the number of communications channels being serviced by the IRB. This model of threading is believed to be supportable by the future release of Nexus- which will adopt a thread manager as part of Globus.

5.3.3 Networking Quality of Service Capabilities

The IRB has API calls to allow an application to specify network quality of service parameters. These API calls are only stubs. Nexus has these capabilities planned in the future. As these capabilities become available they will also be made available to the IRB.

5.3.4 Further Reducing Redundant Memory Copying

Currently when data is transmitted from- or received by- Nexus, a single redundant copy of the data has to be made at each phase. A "raw" TCP program however, does not incur this overhead since no redundant copies are made.

To solve this problem future releases of Nexus will provide direct *put* and *get* operations that will allow Nexus to directly fill the IRB's channel buffers rather than having to make redundant copies of the data.

5.3.5 Security

The IRB prototype possesses no capabilities for maintaining security in keys. Nor does it maintain security in transmitted data. The issues of security are important and complex and have been excluded in the first prototype to simply its design. Future versions of the IRB should provide facilities for selecting secure transmission channels. Nexus will once again be responsible for securing those channels.

5.3.6 Locking Scheme for Keys

The current IRB prototype provides no provisions for locking keys. The kind of locking expected to be useful for Teleimmersion is a non-blocking form of entry consistency. Entry consistency in distributed shared memory circles amounts to the explicit locking and unlocking of potentially shared entities. A non-blocking scheme is needed in Teleimmersion to prevent a realtime application from blocking when a lock is requested. Instead when a lock is requested a callback function is also provided that is called by the IRB once news about the status of the lock arrives. If the news is that the lock has been acquired the application may then safely read the contents of the key and modify it. If the lock could not be acquired the client application is still able to read the key, however any attempted updates to the key will be discarded.

In addition to providing a callback function two time-out parameters are needed. The first specifies how long the application is willing to wait to acquire a lock. The second specifies how long the application is expected to need the lock once it has been acquired. If the application needs to keep its lock it must refresh the lock before the lock expires. This is provided to prevent a client application from inadvertently forgetting to release a lock and hence potentially preventing other participants from ever accessing the data again. In addition it provides a simple deadlock resolution scheme.

5.3.7 Preventing Cycles

A cycle occurs when keys are linked together across multiple IRBs in a loop. When one key is modified the change will propagate to its linked counterpart. That key will propagate its update to its counterpart, and so on until the originating key is reached. This will trigger the key to repeat the entire sequence again.

The IRB has only the most simple provisions for detecting cycles. It prevents redundant links from being created between immediately connected IRBs. There are no provisions for cycles that result from loops with more intermediate IRBs. One way to address this issue is to tag data as “original” or “forwarded.” Original data is data that is originated at an IRB and sent to another IRB. On receipt of the data the IRB may need to forward this data to other subscribing IRBs. Before forwarding this data it is tagged as “forwarded data.” Along with this label the original timestamp of the data is also forwarded. On receipt of the data by each IRB the data’s timestamp is compared to the timestamp of a previous copy of the data at the IRB. If the “new” data’s timestamp is as old as or older than the copy already at the IRB, then the “new” data is ignored and hence not forwarded.

It is clear that a deeper study of the problem is needed to provide a suitable solution- in particular a solution that does not occupy valuable networking resources to achieve.

5.3.8 Database Quality of Service

The IRB currently provides a prototype database. The prototype was developed to offer guidance over the kind of API expected by the IRB (this API is specified in Appendix B). The prototype caches all data in main memory and hence will support small-event and medium-atomic data as long as it all fits in the computer’s physical memory. As physical memory is exhausted the operating system will begin to page the data to virtual memory. This will degrade the performance of the entire application to the point that realtime interaction will be impossible. Hence the prototype's replacement must be able to provide multiple levels of data caching, and it must allow the client/server program to negotiate the kind of database performance needed for its particular application (i.e. a notion of database quality of service.) This non-trivial problem is beyond the scope of this dissertation.

5.3.9 Cross-platform Porting

Currently the IRB prototype has been implemented for Silicon Graphics computers primarily because they are presently the platform of choice for Teleimmersion. However since the IRB contains no graphical capabilities, and bases its networking on Nexus (which has been ported to Sun, IBM AIX, IBM SP2, and HP,) the IRB should also be relatively portable.

5.4 A Comparison of the IRB and HLA/DIS for Data Distribution

As the IRB possesses no graphical capabilities, the comparison between the IRB and DIS/HLA will be made in terms of their respective approaches to data distribution.

DIS (Distributed Interactive Simulation) is being phased out and is being replaced by HLA (High Level Architecture for Simulation) - the Department of Defence's new initiative to provide a broader-based simulation standard that is designed to overcome inadequecies and extend the capabilities of DIS.

HLA is an extremely complex and ambitious standard. It is attempting to be the "silver-bullet" of distributed simulation standards- able to support realtime military simulations as well as R&D, engineering, and analysis domains. Their primary objective however is still largely to create a standard that will allow all their future military simulations to interoperate. This standard is currently based on CORBA- in particular Orbix's implementation of CORBA.

There are some major advantages to using CORBA. The CORBA standard is both programming language- and computer architecture independent. CORBA allows the specification of distributed objects at a very high-level. An object's data and member functions (and hence behavior) can be specified. Multiple inheritances, encapsulation, and polymorphism, as in traditional object-oriented programming, are also supported. CORBA eliminates the need to understand low-level networking protocols. It eliminates the need to manually serialize data and marshal parameters for remote procedure calls. It eliminates the need to know explicitly, on which remote computers an object may reside. All these details are handled transparently by CORBA. Ideally all teleimmersive applications should be built by specifying and connecting high-level CORBA-like objects.

CORBA is however, not the first object-oriented system that has been used in distributed virtual reality. The early version of DIVE(25) used an object database called ISIS. DIVE no longer uses ISIS. The main reason was that the overhead incurred in using such an object system was unacceptable for realtime VR applications. This is not to suggest that object databases are patently unsuitable for Teleimmersion. The value of distributed objects is unquestionable. However the use of the currently available commercial object databases, is questionable. All the conveniences provided by distributed objects also incur a significant performance penalty. Current implementations of CORBA are highly inefficient. They do not scale well to support large numbers of small objects, nor do they support very large objects- that is, objects such as enormous scientific data sets(34). When objects are transacted between clients and servers, multiple redundant copies of object data are performed in the ORBs(33). As the size of the data increases so does the overhead of data copying. Data-copying has been shown to be the ultimate limiting factor in the throughput achievable with the CORBA implementations(33). The number of redundant copies made also vary with ORB implementations. ORBeline for example copies the data almost three times more than Orbix(33).

A document(35) prepared for the US Army Simulation, Training and Instrumentation Command reported that the current implementation of the CORBA-based HLA prototype was not sufficient for DIS-type real-time simulation. In particular: latencies for data communication exceeded the DIS threshold. The overall average latencies for attribute updates were between 400 and 500 milliseconds. Furthermore, attribute updates sent over reliable channels were slower, with average latencies of 800-1200 milliseconds observed.

The CAVERNsoft-IRB approach to this problem has been to introduce a light-weight solution that provides an API that is high-level enough to allow distributed applications to be easily built, but is low-level enough to be able to meet the realtime needs of Teleimmersion.

As the IRB is being built from the ground-up it serves as a testbed to allow researchers in Teleimmersion to investigate, understand, and resolve issues relating to realtime performance, as they arise. For example the HLA performance report(35) noted an anomalous batching effect in the delivery of its attribute updates. That is, the first update received in a batch was the longest delayed from its send time, with the delay declining for each successive update received in the batch. They were unable to identify the cause of this batching behavior as it would require detailed analysis of the CORBA-ORB. This batching could have been occurring from within the CORBA implementation or, as a similar experience with the CAVERNsoft-IRB would suggest, the batching behavior might be in fact, due to the inherent buffering of packets that occurs within the implementation of TCP. Even if this were identified as the cause of HLA's problem, CORBA does not allow the low-level manipulation of networking attributes. This may make the integration of up-and-coming networking quality of service capabilities difficult.

It is anticipated that in order for a CORBA-like interface to be use-able in the context of Teleimmersion, it must be completely re-built from the foundation with new optimizations that take into account quality of service issues in network data delivery and data storage. In essence an IRB must be built before an ORB can be built.

5.5 IRB Performance Benchmarks

The examples in sections 5.1.1 and 5.2 illustrate the ease with which the IRB can be used in a variety of situations. However one important question that arises when working with a system of this complexity is "how much performance is lost in maintaining this level of functionality?"

To answer this question a networking-interface and a database performance experiment was conducted. The networking-interface experiment was performed in order to reveal the performance penalty incurred by using CAVERNsoft as compared to Nexus and raw TCP. The database test was performed to give a general impression of the kind of performance that was currently being provided by Keytool, the IRB's prototype database.

All experiments were conducted on a 175Mhz, 128M RAM, Silicon Graphics Indy, running a pthreads-patched version of IRIX 6.2. Version 1.6-alpha of the IRB was used, which included version 4.1 of Nexus. The Indy was taken off the main ethernet local area network as earlier pre-tests showed noticeable performance fluctuations, even though the tests involved the transmission of data from one program to another on the **same** computer.

5.5.1 Networking Interface Performance Comparison

5.5.1.1 General Experimental Setup

The general goal of the experiment was to compare the performance differences between sending data using the standard UNIX TCP socket library, Nexus, and CAVERNsoft. The calls to the UNIX TCP socket library are similar to Stevens' classic examples(36) using blocking sockets with small-packet-buffering disabled. That is, it represents the typical TCP program most client-server applications would use for communication.

All tests were performed by separate processes all residing on the **same** computer. One program would typically send data to another program and the time difference between sending the packet and receiving the packet would be recorded. The programs were executed on the same computer in order to produce results that are independent of networking medium and so that time synchronization between two computers would not be necessary.

5.5.1.2 Experiment 1 : Comparison of raw TCP, Nexus and CAVERNsoft for small packet sizes below 8192 bytes.

The first experiment was to examine the behavior of the three networking systems for small packet sizes. For each system, 100 packets were sent from one process to another. Each packet was timestamped. On receipt of the packet, the elapsed time was recorded. After 100 samples, the recorded results were written to an output file. The experiment was repeated with packet size increments of 128 bytes until a size of 8192 bytes was reached. The mean of all the experimental cases are plotted in Figure 15.

5.5.1.3 Interpretation of Results

The plots seemed to suggest a linear increase in delivery time as packet size increased. Nexus appeared to impose an overhead of approximately 1 millisecond over raw TCP (Figure 17.) CAVERNsoft appeared to impose an additional 0.6 milliseconds on top of Nexus (Figure 16.)

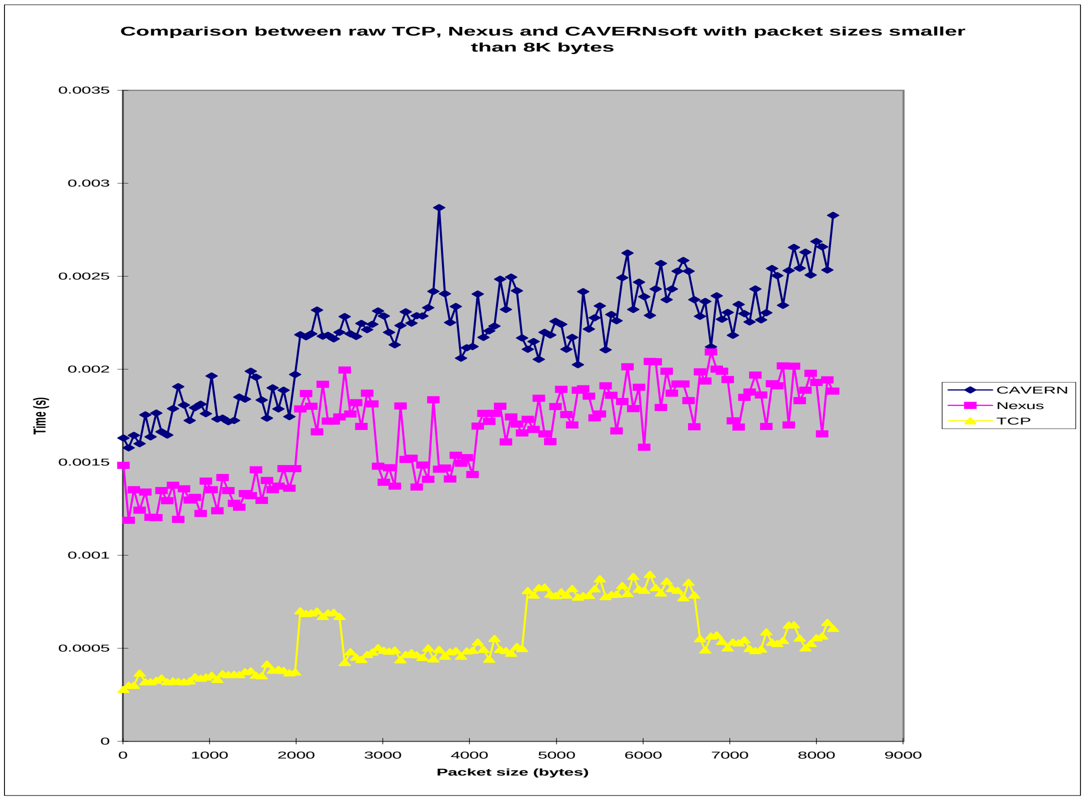


Figure 15. Comparison of raw TCP, Nexus and CAVERNsoft at packet sizes below 8192 bytes.

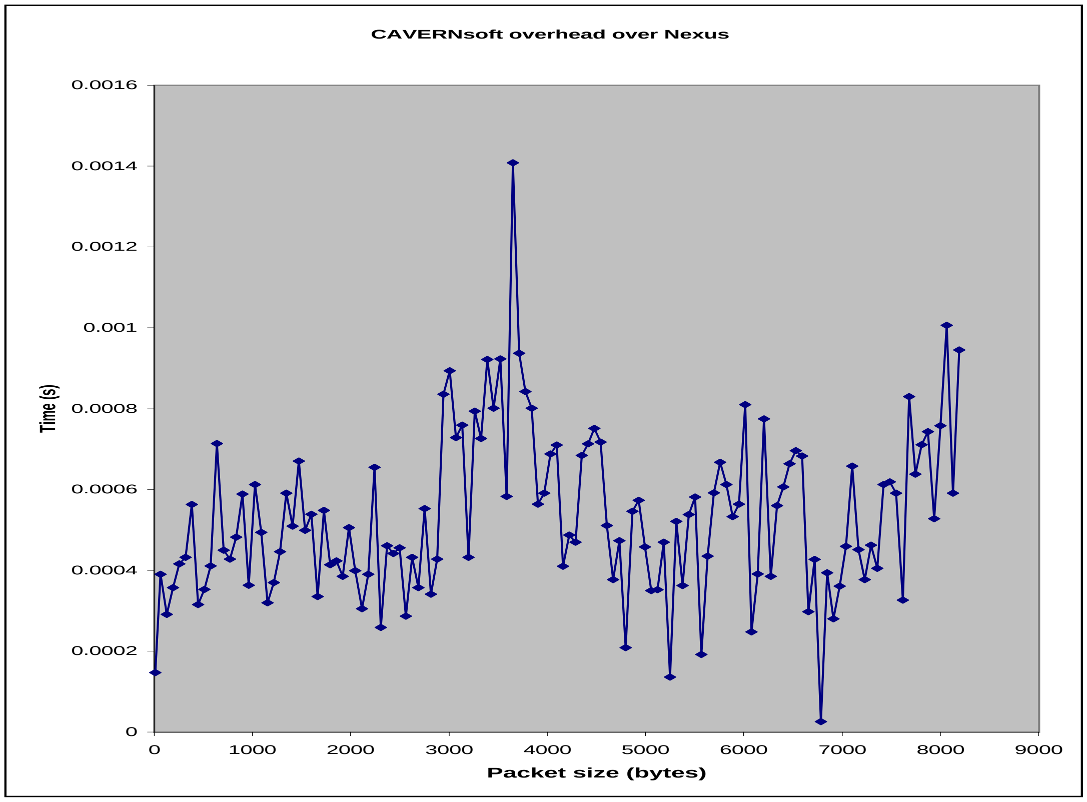


Figure 16. Overhead imposed by CAVERNsoft over Nexus for packet sizes less than 8K bytes.

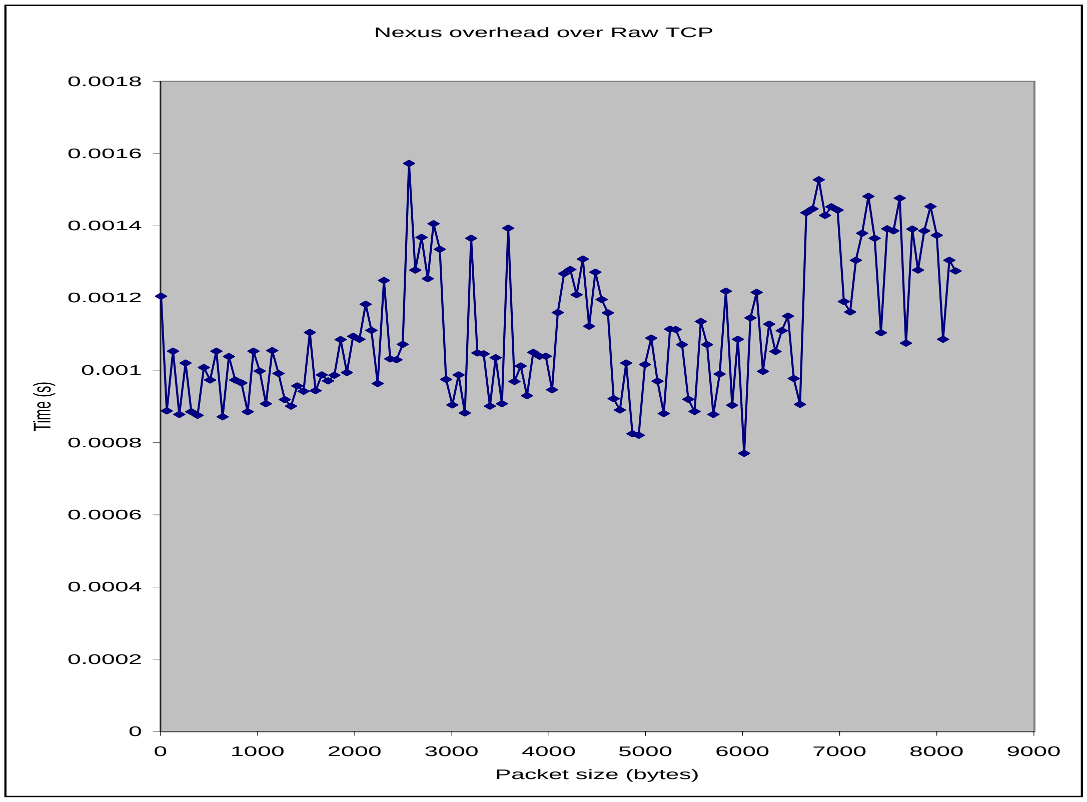


Figure 17. Overhead imposed by Nexus over TCP for packet sizes less than 8K bytes.

5.5.1.4 Experiment 2 : Broad comparison between raw TCP, Nexus and CAVERNsoft.

The second experiment was designed to give a broader performance comparison of the three networking systems. The same setup as in Experiment 1 was used except packet sizes increased in 81920 byte increments until approximately 8 Megabytes was reached.

The mean of all the results are plotted in Figure 18.

5.5.1.5 Interpretation of Results

Figure 18 seems to indicate that CAVERNsoft's performance followed Nexus' closely even as packet size increased. This is further illustrated in a plot of the difference between CAVERNsoft and Nexus (Figure 19.)[1]

Two other differences were observed between the performance of CAVERNsoft/Nexus and raw TCP: a) TCP appeared to significantly out-perform CAVERNsoft/Nexus and b) the TCP graph displayed a "terracing" effect. The latter is believed to be due to the internal buffering of the underlying TCP algorithm. A similar effect was also observed in the benchmarks of HLA(35).

---

[1]As the CAVERNsoft experiments were done independently of the Nexus experiments, there are instances in the graph (negative values) that seemed to indicate that CAVERNsoft out-performed Nexus. This is of course not the case. Initially when this experiment was being designed the thought was to place profiling statements within the CAVERNsoft code so that the overhead of CAVERNsoft and Nexus could be measured at the same time. This was not possible as the program code for CAVERNsoft was so intertwined with Nexus-related calls that considerable sub-measurements would have had to be taken between calls to properly isolate each sub-system's contribution. These sub-measurements would themselves have contributed to an overall performance penalty in CAVERNsoft.

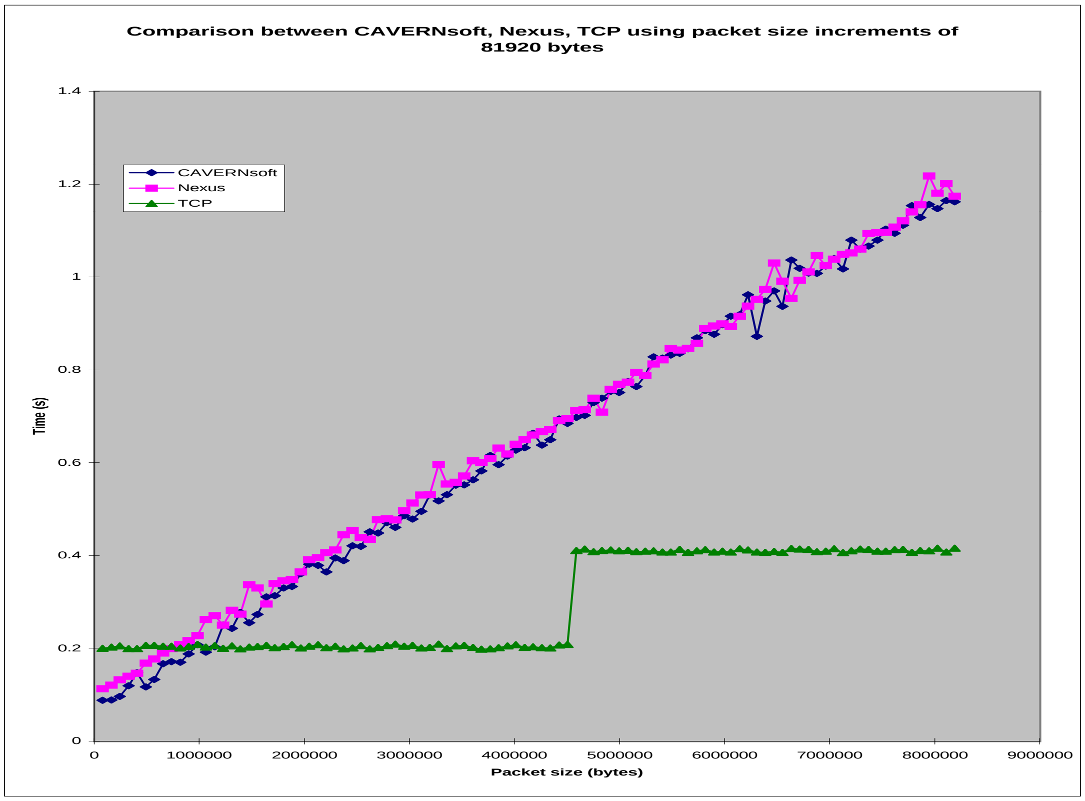


Figure 18. A comparison of raw TCP performance against CAVERNsoft and Nexus using packet size increments of 81920 bytes.

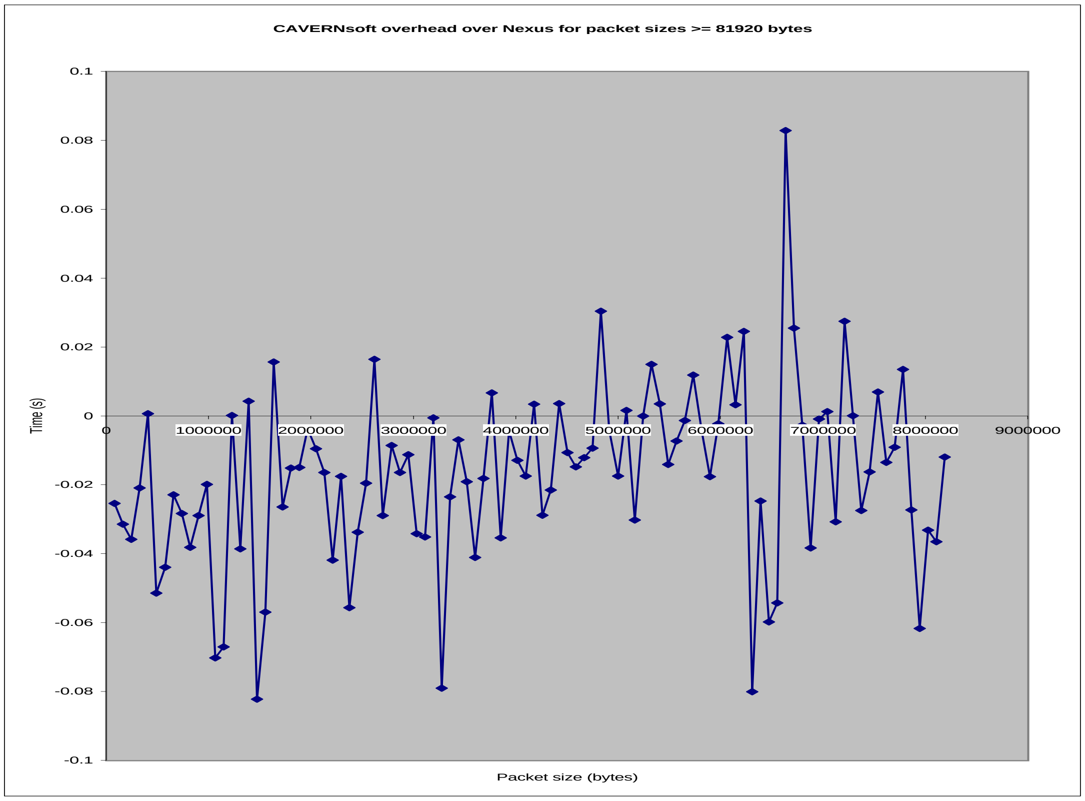


Figure 19. CAVERNsoft overhead over Nexus using packet size increments of 81920 bytes.

The large performance difference between CAVERNsoft/Nexus is believed to be due to the additional redundant memory copies incurred by Nexus during a send and receive of data (as described in section 5.3.4.) To test this hypothesis, the raw TCP experiments were repeated with one redundant copy made at the client and at the server. The results, which are plotted in Figure 20, seemed to confirm this.

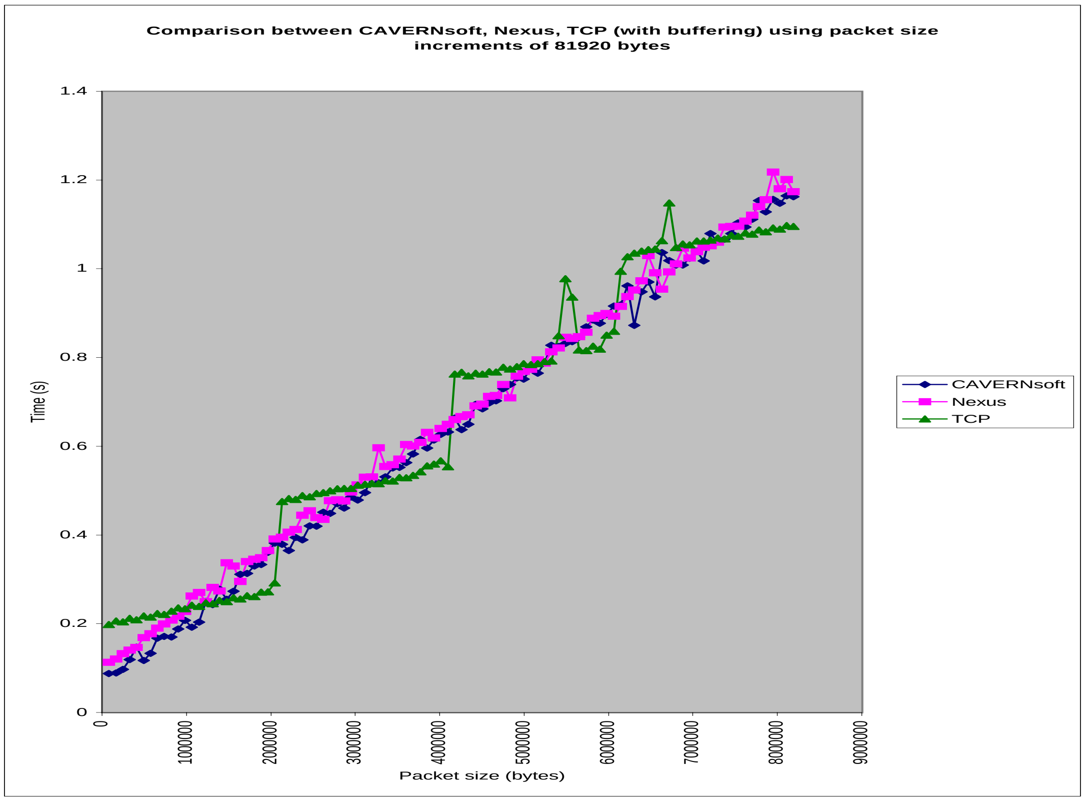


Figure 20. A comparison of buffered TCP performance against CAVERNsoft and Nexus using packet size increments of 81920 bytes.

Despite the current large performance difference between Nexus and raw TCP, future releases of Nexus, with its direct *put* and *get* capabilities, will eliminate this difference. However at the same time it should be noted that although the TCP results showed better overall performance than CAVERNsoft/Nexus the responsiveness of CAVERNsoft/Nexus send calls were higher. Figure 21 shows the elapsed time for performing the send calls for each of the networking interfaces. Note that CAVERNsoft/Nexus returns from the send calls sooner than raw TCP. This is because the copying of the data buffer in Nexus allowed it to perform the send as a separate concurrent thread from the main thread whereas the raw TCP program must block until the entire send is complete. With a strictly non-buffered send, the client application programmer must write his/her own thread to deliver the data concurrently.

5.5.2 Database Performance Experiment

This experiment examines the performance of committing data to the Keytool prototype database as compared to writing the same data to a UNIX file system. Another possible experiment might have been to compare the prototype to existing commercial databases. This comparison was not performed as the IRB's database was implemented only as a placeholder for an eventual, more efficient and robust database. The placeholder's purpose was to help realize the proof of concept IRB and to enable the specification of Keytool's API (see Appendix B).

The database experiment is therefore only to provide some general impression of the kind of performance that is presently being offered by the prototype IRB.

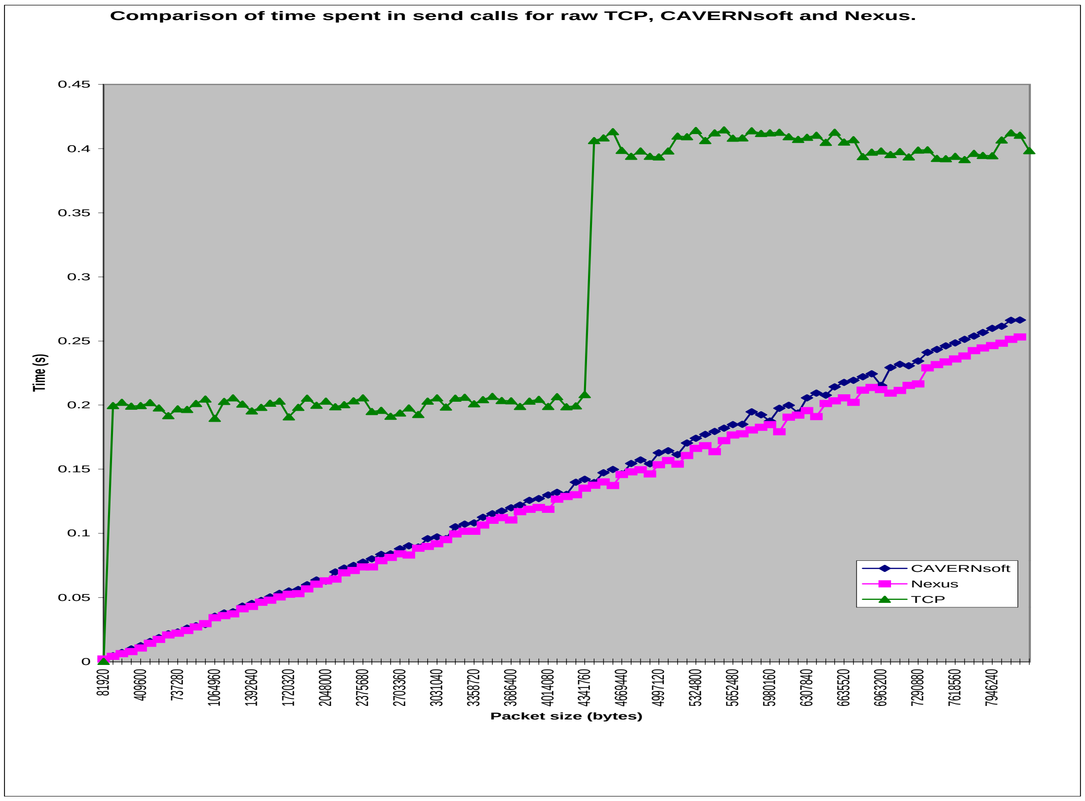


Figure 21. Comparison of time spent in send calls for raw TCP, CAVERNsoft and Nexus for packet size increment of 81920 bytes.

5.5.2.1 General Experimental Setup

In the database experiments, networking was not needed. Hence only a single IRB-based program was launched. This program timed a number of immediately successive commit calls to a single key on the IRB. The experiment began with a database entry of 163840 bytes and 50 trials. Successive experiments increased the database entry size in increments 163840 bytes until 16384000 bytes was reached.

Following this experiment another identical experiment was performed to write data to a single file using the standard UNIX fwrite() system call.

All database commits and file fwrites were performed on a non-NFS-mounted local disk space.

5.5.2.2 Interpretation of Results

The initial findings were somewhat surprising. The plots labeled "CAVERNsoft" and "fwrite single file" in Figure 22 seemed to suggest that Keytool was significantly out-performing raw fwrites. This did not seem correct since Keytool based its own file writes on fwrite.

The reason for this discrepancy was that the simple fault tolerance provision in Keytool was also helping to improve its overall performance. As described in section 4.5, a small degree of fault tolerance was added to Keytool so that commits were done by first writing the data to a temporary file and then renaming the file to replace the older version. This fwrite to a temporary file meant that the IRB process would not have to wait until the previous file I/O operation was completely finished by the operating system before beginning the next fwrite. To

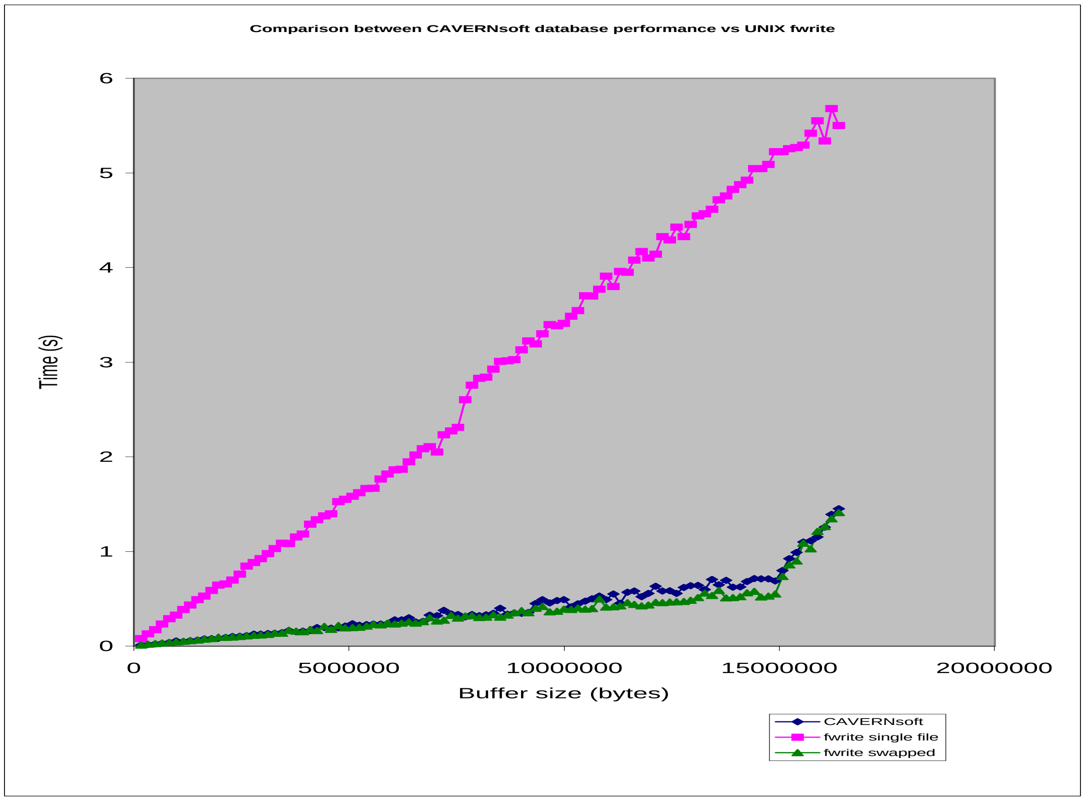


Figure 22. CAVERNsoft database performance vs UNIX fwrite.

verify that this was indeed the cause of the performance improvement, the fwrite test program was modified to perform file writes in a similar manner. The plot in Figure 22 labeled "fwrite swapped" seemed to confirm this. From comparing Keytool with the file-swapped fwrite results, it appears that Keytool still imposes very little additional overhead.

5.5.3 Summary of Findings

In general the prototype version of CAVERNsoft imposes very little additional networking-interface overhead over Nexus. In experiments that involved the transmission of small packets, the imposed overhead was approximately 0.6 milliseconds. In experiments that involved the transmission of large packets, the imposed overhead became too small to differentiate from the time spent in delivering the data. The imposed overhead by Nexus over using low-level UNIX TCP calls was approximately 1 millisecond. For large packets however, raw TCP significantly out-performed Nexus. This was due to the redundant buffering currently present in Nexus' send and receive calls. This buffering delay will be removable in future releases of Nexus. However, there is a trade-off between getting a packet to its destination as quickly as possible versus returning from the packet send call as quickly as possible. In the former case the client's data is directly sent to the destination without any intermediate buffering. Hence the client program will need to block until the data has completely reached its destination before it can perform the next modification to the data. In the latter case the client's data is copied by the IRB so that the IRB may send the copy independently of whatever new modifications the client may make immediately following the send call. This improves concurrency at the expense of performing one redundant memory copy. This trade-off is a common problem faced by all higher-level networking libraries. Future versions of the IRB's API will have to offer both options and allow the programmer to choose the one most appropriate for his/her application.

The database experiments suggested that Keytool imposes very little overhead over raw UNIX file writes. This database is only a placeholder for its eventual replacement. However, even as a prototype it has been found to be useful in storing and retrieving small-event and medium-atomic data- as demonstrated by the NICE and General Motors examples.

All the experiments described above were conducted on a specific configuration of equipment. These experiments will produce different results on different systems. CAVERNsoft's performance will increase on a faster computer. Hence to support time-critical teleimmersive applications that could potentially be running on computers with varying capabilities, it is important that the IRB provide a self-benchmarking and statistics-gathering capability to inform the way it grants networking and database quality of service contracts.

CHAPTER 6

CONCLUSION

A suitable architecture for Teleimmersion should: facilitate the rapid construction of arbitrary distributed topologies; provide support for various networking protocols (including reliable and unreliable, unicast, broadcast and multicast) and quality of service capabilities; provide facilities for supporting concurrent programming (both a message-passing and a distributed shared memory model); and provide support for persistence- where small-event, medium-atomic and large-segmented data can be seamlessly managed.

To support these capabilities, the Information Resource Broker has been proposed, and a prototype has been implemented and evaluated. The IRB provides a layer of software above that of a threaded networking library (such as Nexus and ACE(33)) but below that of a distributed object database (such as CORBA).

The IRB provides application developers with the ability to treat applications symmetrically as both clients and servers, allowing any IRB-based program to link with any other IRB-based program in almost any desired topology, to share information. Benchmarks show that it is possible to provide both client and server capabilities with little additional overhead on top of normal data distribution. The IRB unifies data sharing in a manner that allows developers to conveniently treat data sharing as either message passing or as a distributed shared memory or database.

This initial work can serve as a foundation for many new areas of interesting research in Teleimmersion:

- In order for the IRB to reach its full potential, the ability to control networking and database quality of service must be implemented. Networking QoS is needed to ensure that data streams are "smooth-enough" for the application at hand. Database QoS is needed to ensure that the application can store, possibly streams of a teleimmersive recording session, to the adequate degree of reliability and performance. To inform the IRB's ability to provide certain levels of QoS, the IRBs must contain some means of self-profiling.
- The building of continuously persistent LIMBO and DOMAIN spaces using the CAVERNsoft architecture will be the primary means to motivate the development of new teleimmersive applications. The proper integration of technology in these spaces will require both the careful design and modularization of the components being integrated (so that the components can be used individually to retro-fit non-teleimmersive applications,) and the consideration of the human-factors aspects of working in the collaborative space.
- The problems involved in supporting state persistence will be an exciting and important area of Teleimmersion research. State persistence involves both the support of asynchronous collaboration where the entire state of the world may need to be saved; and the support of continuous recordings of the world as it evolves. Research obviously needs to be done in the area of efficient realtime storage and playback of the recordings- hence the need to provide QoS capabilities in the IRB. However research also needs to be done in

supporting indexing and querying of the recorded data. Hence human-factors research needs to be done in providing sensible interfaces with which collaborating participants can navigate in virtual time as well as in virtual space.

APPENDICES

Appendix A

THE CAVERNSOFT APPLICATION PROGRAMMER'S INTERFACE

Contents

Appendix A (Continued)

1

CAVERN_irb_c* **CAVERNInit** (int *argc, char*** argv, CAVERN_initAttrib_c *initAttr=NULL)

This is the first thing you do to start CAVERN

This is the first thing you do to start CAVERN.

Sample call:

```
CAVERN_irb_c *personalIRB = CAVERNInit(&argc, &argv, NULL);
```

Once it is called it will spawn off a number of threads to support your personal IRB. Every CAVERN client has a personal IRB. The personal IRB is the main object you use to control CAVERN. See the CAVERN_irb_c class for a description of what this handle can do.

This call returns a handle to the personal IRB or NULL if the initialization failed.

CAVERNsoft always creates a local default data-store in which it can store persistent data. The default data-store is created in a directory called CAVERN_DEFAULT_DB in the same location where your CAVERNsoft-based application is executed. The name of the default data-store directory can be changed by setting one of the options in the CAVERN_initAttrib_c class as an argument to CAVERNInit().

WARNING 1: CAVERN uses threads. Therefore it is important that CAVERNInit() is called after the LAST fork in your program. For example, in the CAVE you call CAVERNInit() just before the CAVE's main process' while loop. This is so that the threads belong to the main process of the CAVE. What this also means is that CAVERN calls should ONLY be made in the CAVE main process and NOT the draw processes.

If someday the CAVE library gets rewritten with threads then we will all be one happy family.

WARNING 2: CAVERN's networking is supported by Nexus which creates a thread every time a new message arrives from a remote source. Currently if you have too many messages coming in simultaneously Nexus will spawn so many threads that eventually it will run out of system resources and terminate your program. This will be fixed in the future when Nexus modifies its threading scheme. But in the meantime one way to reduce this problem is to try to not send enormous numbers of small packets in a small amount of time (like tracker packets)-

separate them with a small sginap(3). This is one of the reasons why all the demos in this CAVERN distribution include small delays after each send of data. The other reason is to slow down output so that you can read it.

Parameters:

`initAttr` — The attribute sets any special conditions for initializing CAVERN (like port number.) If it is set to NULL then default settings are assumed. Also if this is set to NULL, CAVERN will look for the CAVERN_PORT environment variable for a possible starting port number.

`argc` — Argc from main(argc,argv).

`argv` — Argv from main(argc,argv).

2

class **CAVERN_initAttrib_c**

Class to set CAVERN initialization attributes

A.0.3.0.1 Public Members

2.1	void	**setPort** (unsigned short thePort)	*Change CAVERN default main port to something else*	178
	unsigned short	**getPort** ()	*Get default main port of CAVERN client.*	
	void	**setDBName** (char *dbRootName)	*Set the data store name. By default the datastore name is set as: CAVERN_DEFAULT_DB.*	
	char*	**getDBName** ()	*Get the data store name.*	

Class to set CAVERN initialization attributes. With this you can choose the port to open for your personal IRB and you can choose the name of the local CAVERN data-store. Other attributes will be added with time.

2.1

void **setPort** (unsigned short thePort)

Change CAVERN default main port to something else

Change CAVERN default main port to something else. Must be a number greater than 6000. The main port is the port that all IRBs connect to first before establishing any other communication lines.

3

class **CAVERN_linkAttrib_c**

Class for specifying the characteristics of links

A.0.3.0.2 Public Members

3.1	enum	**CAVERN_update_t**	*Choose between ACTIVE or PASSIVE update of linked keys.*	111
3.2	enum	**CAVERN_synch_t**	*Choose between the types of synchronization across the links.*	112
3.3	void	**setInitialSynchType** (CAVERN_synch_t t)	*This to set how data is initially synchronized between 2 keys when a link is first established*	113
3.4	void	**setSubsequentSynchType** (CAVERN_synch_t t)	*This is to set how data is subsequently synchronized between 2 keys in a link*	113
	CAVERN_synch_t	**getSubsequentSynchType** ()	*Returns the current synch type.*	
3.5	void	**setUpdateType** (CAVERN_update_t t)	*Set the link's update type to either active or passive*	114
	CAVERN_update_t	**getUpdateType** ()	*Get update type*	
	CAVERN_synch_t			

getInitialSynchType ()
Get initial synch type.

Class for specifying the characteristics of links. By default links are auto-synchronizing and use active updating. That is links will compare local and remote timestamps to determine if data transfer is needed. Also when data is changed at any site, it is automatically propagated to remote sites.

3.1

enum **CAVERN_update_t**

Choose between ACTIVE or PASSIVE update of linked keys.

A.0.3.0.3 Members

3.1.1	**ACTIVE**	*Active update means key data are synchronized whenever the data changes* ..	112
	PASSIVE	*Passive update means the user must explicitly deliver or request data.*	

3.1.1

ACTIVE

Active update means key data are synchronized whenever the data changes

Active update means key data are synchronized whenever the data changes. This is similar to a server push.

3.2

enum **CAVERN_synch_t**

Choose between the types of synchronization across the links.

A.0.3.0.4 Members

NONE	*Perform no synchronization at all. Useful for initial synch conditions.*
AUTO_SYNCH	*Automatically allow CAVERNsoft to manage synchronization based on timestamps.*
SYNCH_LOCAL_TO_REMOTE	*Synchronize by tranfering data from local to remote site based on time stamp.*
SYNCH_REMOTE_TO_LOCAL	*Synchronize by tranfering data from remote to local site based on time stamp.*
FORCE_LOCAL_TO_REMOTE	*Force tranfer of data from local to remote site regardless of timestamp.*
FORCE_REMOTE_TO_LOCAL	

Force tranfer of data from remote to local site regardless of timestamp.

3.3

void **setInitialSynchType** (CAVERN_synch_t t)

This to set how data is initially synchronized between 2 keys when a link is first established

This to set how data is initially synchronized between 2 keys when a link is first established. When either SYNCH_LOCAL_TO_REMOTE, SYNCH_REMOTE_TO_LOCAL or AUTO are used local and remote timestamps are checked to decide whether data transfer is necessary. In the future this will be extended to also compare checksums since timestamp alone is not a sufficient indication of whether two files are the same or not. For example if 2 keys have different timestamps but are exactly the same then synching is a waste especially if the keys hold very large amounts of data.

3.4

void **setSubsequentSynchType** (CAVERN_synch_t t)

This is to set how data is subsequently synchronized between 2 keys in a link

This is to set how data is subsequently synchronized between 2 keys in a link. In subsequent synch timestamps are not checked. So SYNCH_LOCAL_TO_REMOTE is the same as FORCE_LOCAL_TO_REMOTE. Similarly SYNCH_REMOTE_TO_LOCAL is the same as FORCE_REMOTE_TO_LOCAL. Note. If you use REMOTE_TO_LOCAL synching locally, any changes that you make to your local key will not be propagated to the remote subscriber/client since you have forced data transfer into being unidirectional. These rules can be overriden on temporary bases with: CAVERN_irbLink_c::requestRemote()

3.5

void **setUpdateType** (CAVERN_update_t t)

Set the link's update type to either active or passive

Set the link's update type to either active or passive. Active update means key data are synchronized whenever the data changes. In some sense this is similar to a server push. Passive update means the user must explicitly deliver or request data.

4

class **CAVERN_irbChannel_c**

IRB Communications Channel class

A.0.3.0.5 Public Members

4.1	enum	**status_t**	*Status codes returned from CAVERN_irbChannel_c methods.*	116
4.2	enum	**CAVERN_channelEvent_t**	*Trigger event types for channels.*	117
4.3	enum	**CAVERN_networkReliability_t**	*Network reliability types*	117
4.4	enum	**CAVERN_update_t**	*Update types*	118
4.5		**˜CAVERN_irbChannel_c** ()	*Deleting this channel will close the channel*	118
4.6	void	**open** (CAVERN_irbId_c* remoteIRB, CAVERN_qosAttrib_c* qosAttr, CAVERN_networkReliability_t reliability, CAVERN_irbChannel_c::status_t *retStatus)	*Open the channel to a remote IRB*	119
	void	**close** ()	*Close an open channel.*	
4.7	void	**reserveQoS** (CAVERN_qosAttrib_c *, CAVERN_irbChannel_c::status_t *retStatus)	*Reserve Network Channel's Quality of Service*	120
4.8	CAVERN_irbLink_c*			

link (CAVERN_irbKey_c* localKey,

CAVERN_irbKeyId_c* remoteKeyID,

CAVERN_linkAttrib_c* attr=NULL)

Link a local and remote key together using this channel 121

4.9 void **trigger** (void (*callback)(CAVERN_irbChannel_c::CAVERN_channelEvent_t event, CAVERN_irbChannel_c *thisChannel, void* userData), void* userData)

Trigger on events about the channel ... 121

IRB Communications Channel class. Channels are the actual communication pathways that are set up between local and remote CAVERNsoft-based clients/servers. These channels can be set to transmit data via reliable TCP or unreliable UDP. Also at some point in the future channels can be used to set the kind of quality of service options (bandwidth, latency, jitter requirements) desired by your application. Your application can open as many channels as you like, each possibly with differing communications characteristics. Once these channels are created you may choose what pieces of data are transmitted over them by linking local keys with remote keys (see CAVERN_irbLink_c).

4.1

enum **status_t**

Status codes returned from CAVERN_irbChannel_c methods.

A.0.3.0.6 Members

OK	*Operation succeeded.*
FAILED	*Operation failed.*
NEGOTIATED	*Used for QoS- operation negotiated a lower QoS.*

4.2

enum **CAVERN_channelEvent_t**

Trigger event types for channels.

A.0.3.0.7 Members

BROKEN_CHANNEL	*Broken link due to break of connection.*
QOS_DEVIATION	*Triggers when the QoS of the network service begins to deviate.*

4.3

enum **CAVERN_networkReliability_t**

Network reliability types

A.0.3.0.8 Members

RELIABLE	*Use reliable protocols- DEFAULT.*
UNRELIABLE	*Use unreliable protocols.*

4.4

enum **CAVERN_update_t**

Update types

A.0.3.0.9 Members

ACTIVE_UPDATE	*Active update- DEFAULT.*
PASSIVE_UPDATE	*Passive update.*

4.5

~CAVERN_irbChannel_c ()

Deleting this channel will close the channel

Deleting this channel will close the channel. Also it will unlink all links.

4.6

void **open** (CAVERN_irbId_c* remoteIRB, CAVERN_qosAttrib_c* qosAttr, CAVERN_networkReliability_t reliability, CAVERN_irbChannel_c::status_t *retStatus)

Open the channel to a remote IRB

Parameters:

`CAVERN_irbId_c` — Specifies remote IRB.

`CAVERN_qosAttrib_c` — Specifies the desired QoS. It returns set with the QoS it was able to negotiate. Currently this feature has not been implemented since Nexus does not yet support QoS capabilities.

`CAVERN_networkReliability_t` — selects either CAVERN_irbChannel_c::RELIABLE or CAVERN_irbChannel_c::UNRELIABLE transmission.

`status` — returns OK if got contract ; NEGOTIATED if had to negotiate for lower QoS; FAILED if completely failed.

4.7

void **reserveQoS** (CAVERN_qosAttrib_c *, CAVERN_irbChannel_c::status_t *retStatus)

Reserve Network Channel's Quality of Service

Parameters:

`CAVERN_qosAttrib_c` — Specifies the desired network quality of service (QoS). It returns set with the QoS it was able to negotiate. If no QoS attributes are desired, then set this parameter to NULL.

`retStatus` — returns OK if got contract ; NEGOTIATED if had to negotiate for lower ; FAILED if completely failed. Currently this feature has not been implemented since Nexus does not yet support QoS capabilities.

4.8

CAVERN_irbLink_c* **link** (CAVERN_irbKey_c* localKey, CAVERN_irbKeyId_c* remoteKeyID, CAVERN_linkAttrib_c* attr=NULL)

Link a local and remote key together using this channel

Link a local and remote key together using this channel. Returns a link object if successful, else NULL. Linking allows you to store data in a local key and have it automatically propagated to a remotely linked key and vice versa. Linking to a non-existant remote key will dynamically create the remote key.

4.9

void **trigger** (void (*callback)(CAVERN_irbChannel_c::CAVERN_channelEvent_t event, CAVERN_irbChannel_c *thisChannel, void* userData), void* userData)

Trigger on events about the channel

Trigger on events about the channel. For example, if a connection breaks the user can be warned by a BROKEN_CHANNEL event.

Parameters: `callback` — Callback should be of the form: void callback(CAVERN_irbChannel_c::CAVERN_channelEvent_t event, CAVERN_irbChannel_c *thisChannel, void* userData);

5

class **CAVERN_irbId_c**

Class to specify a remote IRB to connect

A.0.3.0.10 Public Members

void	**setAddress** (char *ipAddr)	*Set address of remote IRB.*
void	**setPort** (unsigned short prt = CAVERN_IRB_DEFAULT_MAIN_PORT)	*Set port of remote IRB.*
char*	**getAddress** ()	*Get address.*
unsigned short	**getPort** ()	*Get Port.*

Class to specify a remote IRB to connect. For now this is identified by IP and Port. In the future we can use Globus to do it.

6

class **CAVERN_irbKeyId_c**

Class to specify a key's ID

A.0.3.0.11 Public Members

	char*	**getPath** ()	*Get the designated path of key.*	
	char*	**getName** ()	*Get text name of key.*	
6.1	void	**setPath** (char*)	*Set the designated path of key*	123
	void	**setName** (char*)	*Set text name of key.*	
	char*	**getFullName** ()	*Get full combined path/name*	

Class to specify a key's ID. A key is identified by a path name and a name. Path's can be considered to correspond to a UNIX directory hierarchy rooted at CAVERN's default database directory (usually CAVERN_DEFAULT_DB). If the path is not set or is set to NULL then the root database directory path is assumed.

6.1

void **setPath** (char*)

Set the designated path of key

Set the designated path of key. e.g. setPath("foo/bar");

7

class **CAVERN_irbKey_c**

CAVERN IRB Key class

A.0.3.0.12 Public Members

Appendix A (Continued)

Commit the local key to persistent store 135

7.14 void **deliver** (status_t *status, CAVERN_irbKey_c::CAVERN_irbKeyBlocking_t blockingType = CAVERN_irbKey_c::NON_BLOCKING)
Dispatch data inside current key 135

7.15 void **deliverIncludingPassive** (status_t *status, CAVERN_irbKey_c::CAVERN_irbKeyBlocking_t blockingType = CAVERN_irbKey_c::NON_BLOCKING)
This does the same thing as deliver except it delivers to passive links as well 136

7.16 void **trigger** (void (*callback)(CAVERN_irbKey_c::CAVERN_irbKeyEvent_t event, CAVERN_irbKey_c *thisKey, void* userData), void* userData)
Trigger on an event 137

CAVERN_irbKeyId_c **getKeyId** () *Get a copy of the key Id that defined this key.*

CAVERN IRB Key class. Objects of this type are instantiated by calling CAVERN_irb_c::define(). When you are done refering to the key delete it to free up system resources.

7.1

~CAVERN_irbKey_c ()

The destructor will dereference the local key so that it may be garbage collected to free up CAVERN's resources

The destructor will dereference the local key so that it may be garbage collected to free up CAVERN's resources. Hence it is a way to undefine a key. There is currently no call to undefine a key permanently including the persistent store.

7.2

enum **status_t**

Status codes returned from CAVERN_irbKey_c methods.

A.0.3.0.13 Members

OK	*Operation succeeded.*
FAILED	*Operation failed.*
KEY_STALE	*Access to the key failed because the key is no longer valid; delete the key and redefine it if you wish to access it*
BUFFER_TOO_SMALL	

Returned if you attempt a get() but did not provide a large enough buffer for CAVERNsoft to store the data into

7.3

enum **CAVERN_irbKeyBlocking_t**

Used to specify if specific key operations should block till completion or not (See put(), deliver(), deliverIncludingPassive())

A.0.3.0.14 Members

BLOCKING	*Blocking.*
NON_BLOCKING	*Non-blocking.*

7.4

enum **CAVERN_irbKeyEvent_t**

Trigger event types for keys.

A.0.3.0.15 Members

NEW_DATA_ARRIVED

New data has arrived at the key.

7.5

void **put** (char *buffer, int *sze, status_t *status, CAVERN_irbKey_c::CAVERN_irbKeyBlocking_t blockingType = CAVERN_irbKey_c::NON_BLOCKING)

Put data into key

Put data into key. This replaces the previous contents with the new contents. This call usually results in a copy of the entire data to the remote site.

Parameters:

`buffer` — gets copied to key.

`sze` — amount of data to copy. Returns sze successfully copied.

`status` — returns status: FAIL occurs if any general problems occurred (like internal mem allocation) KEY_STALE occurs if this key is no longer valid for data access. OK is returned if the put is successful.

`blockingType` — if set to BLOCKING will return only after the data has been delivered, otherwise this function will return immediately after having initiated delivery. Blocking is ususually used if you don't wish to continue with the flow of your program until a piece of data has been guaranteed to be delivered to the destination. For example you would perform a blocking put before you did a remoteCommit on a link.

7.6

void **import** (char *filename, int *sze, status_t *status, CAVERN_irbKey_c::CAVERN_irbKeyBlocking_t blockingType = CAVERN_irbKey_c::NON_BLOCKING)

Import data into key from a file

Import data into key from a file. This replaces the previous contents with the new contents. This call usually results in a copy of the entire data to the remote site.

Parameters:

`filename` — specifies the file to load.

`sze` — this returns the size of the file loaded.

`status` — returns status: FAIL occurs if any general problems occurred (like internal mem allocation) KEY_STALE occurs if this key is no longer valid for data access. OK is returned if the put is successful.

`blockingType` — if set to BLOCKING will return only after the data has been delivered, otherwise this function will return immediately after having initiated delivery. Blocking is ususually used if you don't wish to continue with the flow of your program until a piece of data has been guaranteed to be delivered to the destination. For example you would perform a blocking put before you did a remoteCommit on a link.

7.7

void **export** (char *filename, status_t *status)

Export a key to an output file

Parameters:

`status` — returns either OK or FAILED. KEY_STALE is returned if this key is no longer valid for data access.

`filename` — is output file to export to.

7.8

void **get** (char *bufferToFill, int *sze, status_t *status)

Get a copy of the data from key

Parameters:

`bufferToFill` — is a user provided buffer into which the data is copied.

`sze` — is the size of the data buffer provided. Sze returns the size of the data copied into the buffer.

`status` — returns status: BUFFER_TOO_SMALL occurs when the data in the key is larger than data buffer provided by user. OK if everything was fine. KEY_STALE if for some reason this key is no longer active.

7.9

void **getAutoAlloc** (char *&autoAllocatedBuffer, int *sze, status_t *status)

Get a copy of the data in the key but auto allocate memory for the buffer

Get a copy of the data in the key but auto allocate memory for the buffer. I.e. you give it a plain pointer and it will return a chunk of memory. The returned buffer must be manually deleted by you. If memory could not be allocated the buffer returns NULL, the size returns 0.

Parameters: `status` — returns OK if everything was fine. Returns FAILED if a memory allocation problem occurred. Returns KEY_STALE if the this key is no longer valid.

7.10

void **getDirect** (char *&direct, int *sze, status_t *status)

Get the direct pointer to the data in the key

Get the direct pointer to the data in the key. Use this with CAUTION. This forces a lock on CAVERNsoft's internal database so it could potentially prevent new incoming data from arriving. Use the data and release the resource with releaseDirect() as quickly as possible.

Parameters: `status` — returns OK or KEY_STALE.

See Also: releaseDirect()

Appendix A (Continued)

7.11

void **allocDirect** (char *&direct, int *sze, status_t *status)

Similar to getDirect

Similar to getDirect. This function allocates memory in the key of a size specified by the sze parameter. Sze also returns the sze of the data successfully allocated. Direct is a pointer to the allocated memory. Use this with CAUTION. This forces a lock on CAVERNsoft's internal database so it could potentially prevent new incoming data from arriving. Use the data and release the resource with releaseDirect() as quickly as possible.

Typical use of this function is:

aKey->allocDirect(buffer,....);

fillBufferWithData(buffer);

aKey->releaseDirect();

aKey->deliver(......);

Warning. There is currently nothing that guarantees that the data in the key won't be updated by new incoming data between the time of releaseDirect() and deliver().

Parameters:	`status` — returns OK, KEY_STALE or FAILED
See Also:	releaseDirect()

7.12

void **releaseDirect** (status_t *status)

Release the pointer to the data in the key

Release the pointer to the data in the key. See getDirect(). At the present time, multiple successive calls to releaseDirect can produce undefined behavior.

Parameters: `status` — returns either OK or KEY_STALE.

7.13

void **commit** (status_t *status)

Commit the local key to persistent store

Commit the local key to persistent store. If you wish to commit a remote key to persistent store see CAVERN_irbLink_c::commitRemote().

7.14

void **deliver** (status_t *status, CAVERN_irbKey_c::CAVERN_irbKeyBlocking_t blockingType = CAVERN_irbKey_c::NON_BLOCKING)

Dispatch data inside current key

Dispatch data inside current key. It ensures that if this key is participating in a link it will dispatch its changes to all the remote connections. This is good to do after data is modified with getDirect and releaseDirect since these calls have no knowledge of whether you have modified the key or not.

Delivery abides by the constraints set by CAVERN_linkAttrib_c during the creation of the link.

See CAVERN_irbLink_c and CAVERN_linkAttrib_c.

Parameters: `status` — returns either OK or KEY_STALE.

`blockingType` — if set to BLOCKING will return only after the data has been delivered, otherwise this function will return immediately after having initiated delivery. Blocking is ususually used if you don't wish to continue with the flow of your program until a piece of data has been guaranteed to be delivered to the destination. For example you would perform a blocking put before you did a remoteCommit on a link.

7.15

void **deliverIncludingPassive** (status_t *status, CAVERN_irbKey_c::CAVERN_irbKeyBlocking_t blockingType = CAVERN_irbKey_c::NON_BLOCKING)

This does the same thing as deliver except it delivers to passive links as well

This does the same thing as deliver except it delivers to passive links as well. See CAVERN_irbLink_c.

Parameters: `status` — returns either OK or KEY_STALE.

`blockingType` — if set to BLOCKING will return only after the data has been delivered, otherwise this function will return immediately after having initiated delivery. Blocking is ususually used if you don't wish to continue with the flow of your program until a piece of data has been guaranteed to be delivered to the destination. For example you would perform a blocking put before you did a remoteCommit on a link.

7.16

void **trigger** (void (*callback)(CAVERN_irbKey_c::CAVERN_irbKeyEvent_t event, CAVERN_irbKey_c *thisKey, void* userData), void* userData)

Trigger on an event

Trigger on an event. This will call a user-provided callback when an event occurs. See CAVERN_irbKeyEvent_t. WARNING: Currently you cannot perform a put() or any of the deliver() calls on this same key while you are in the user's callback [get() calls are ok.] This is because the key is locked to protect other clients from writing to your key while you are reading it. Hopefully soon we will come up with a solution for this.

Parameters:

`callback` — is the callback to call when trigger occurs. Callback is of the form: void myCallback(CAVERN_irbKey_c::CAVERN_irbKeyEvent_t event, CAVERN_irbKey_c *thisKey, void* userData);

`userData` — is address of data to pass into callback.

8

class **CAVERN_irbLink_c**

Link management class

A.0.3.0.16 Public Members

8.1 enum **status_t** *Status codes returned from CAVERN_irbLink_c member functions.* 140

8.2 enum **CAVERN_linkBlocking_t** *Used to set either blocking or non-blocking remote request.* 141

8.3 enum **CAVERN_linkEvent_t** *Trigger event types for links.* 141

8.4 void **trigger** (void (*callback)(CAVERN_irbLink_c::CAVERN_linkEvent_t event, CAVERN_irbLink_c *thisLink, CAVERN_irbLink_c::status_t status, void* userData), void* userData) *Set a user-specified callback to be called when an event occurs* 142

˜CAVERN_irbLink_c () *This destructor will also unlink the link.*

8.5 void **commitRemote** (status_t *status)

Tell the remote IRB to commit the remote key's data to persistent data store 143

8.6 enum **CAVERN_requestType_t**
Remote request types. 143

8.7 void **requestRemote** (CAVERN_requestType_t requestType, status_t *status, CAVERN_irbLink_c::CAVERN_linkBlocking_t blocking = CAVERN_irbLink_c::NON_BLOCKING, int timeOutTime = 30)
Request the data from a remote source over a specific link 144

Link management class. This class allows you to manage a link. It allows you to commit the data in a remote key to a persistent data-store. It allows you to explicitly request data from a remote key rather than having to wait for data to be automatically propagated to you. More features will be added with time.

8.1

enum **status_t**

Status codes returned from CAVERN_irbLink_c member functions.

A.0.3.0.17 Members

FAILED	*Operation failed*
OK	*Operation succeeded*
TIMED_OUT	*Operation timed out*
NO_DOWNLOAD_NEEDED	*This is a status message returned from a remote request if the synchronizing time stamps determine that a remote download was not necessary.*

8.2

enum **CAVERN_linkBlocking_t**

Used to set either blocking or non-blocking remote request.

A.0.3.0.18 Members

NON_BLOCKING	*Non blocking remote request.*
BLOCKING	*Blocking remote request.*

8.3

enum **CAVERN_linkEvent_t**

Trigger event types for links.

A.0.3.0.19 Members

REMOTE_REQUEST *This event occurs when there is news about a remote request.*

8.4

void **trigger** (void (*callback)(CAVERN_irbLink_c::CAVERN_linkEvent_t event, CAVERN_irbLink_c *thisLink, CAVERN_irbLink_c::status_t status, void* userData), void* userData)

Set a user-specified callback to be called when an event occurs

Set a user-specified callback to be called when an event occurs. Callback should be of the form:

void callback(CAVERN_irbLink_c::CAVERN_linkEvent_t event, CAVERN_irbLink_c *thisLink, CAVERN_irbLink_c::status_t status, void* userData);

This callback may receive for e.g. a REMOTE_REQUEST event with a status of TIMED_OUT.

Parameters: `userData` — is for passing any user data to the callback on invocation.

8.5

void **commitRemote** (status_t *status)

Tell the remote IRB to commit the remote key's data to persistent data store

Parameters: `status` — returns either FAILED or OK.

8.6

enum **CAVERN_requestType_t**

Remote request types.

A.0.3.0.20 Members

DEFAULT	*Default request will compare timestamps.*
FORCED_REQUEST	*Forced request will always download data regardless of timestamps.*

8.7

void **requestRemote** (CAVERN_requestType_t requestType, status_t *status, CAVERN_irbLink_c::CAVERN_linkBlocking_t blocking = CAVERN_irbLink_c::NON_BLOCKING, int timeOutTime = 30)

Request the data from a remote source over a specific link

Request the data from a remote source over a specific link. Used primarily for passive updates. Callback of remote request could get status: FAILED, NO_DOWNLOAD_NEEDED, TIMED_OUT or OK. OK event means data will be transmitted to your locally linked key. This will fire the irb Key's callback for new incoming data.

Parameters: `timeOutTime` — is the time (in seconds) this request waits for either status information or incoming data. When timeout occurs a TIME_OUT event will fire.

`blocking` — is used to make this request a blocking or non-blocking request. If non-blocking then this function will return immediately with either a status of OK or FAILED that reflects the status of calling this request (ie not the status of the result of the completed request). The callback will actually reveal the true status of the completed request. If blocking then this function waits till remote status information arrives and remote data is completely downloaded (whichever occurs first). The status in this case would be the status of the completed request- i.e. the status the callback would get.

9 class **CAVERN_irbMcastChannel_c**

IRB Multicast Communications Channel class

A.0.3.0.21 Public Members

IRB Multicast Communications Channel class. A multicast channel opens up to a multicast group rather than to a remote IRB. Multicast channels use the standard unreliable multicast protocol. If you wish to multicast between distantly located remote sites you will either need your system administrator to set up a multicast tunnel, or you need to create your own version of a tunnel by forming explict CAVERN_irbChannel_c's and links to the remote site. Note: when you open a connection to a multicast group on a machine you cannot open a connection another multicast group from the same machine.

9.1

enum **status_t**

Status codes returned from CAVERN_irbChannel_c methods.

A.0.3.0.22 Members

OK	*Operation succeeded.*
FAILED	*Operation failed.*
NEGOTIATED	*Used for QoS- operation negotiated a lower QoS.*

9.2

void **open** (char * mcastAddr, unsigned short mcastPort, int mcastTTL, CAVERN_qosAttrib_c* qosAttr, CAVERN_irbMcastChannel_c::status_t *retStatus)

Open the multicast channel to a multicast group

Open the multicast channel to a multicast group. All IRB programs expected to receive the same packets must listen to the same multicast address and port.

Parameters:

`mcastAddr` — Specifies a multicast address as a string. Multicast addresses should be of the form 224.x.x.x

`mcastPort` — Specifies a multicast port.

`CAVERN_qosAttrib_c` — Specifies the desired QoS. It returns set with the QoS it was able to negotiate. Currently this feature has not been implemented since Nexus does not yet support QoS capabilities.

`status` — returns OK if got contract; NEGOTIATED if had to negotiate for lower QoS; FAILED if completely failed.

9.3

CAVERN_irbMcastLink_c* **link** (CAVERN_irbKey_c* localKey)

Link a local key to the multicast channel

Link a local key to the multicast channel. Note any remote IRBs listening to the same channel with a key that is linked with the SAME NAME will receive data from this key if this key is modified. Conversely if any remote IRB modifies the key this key will also be updated. Hence multicast is like a big pool of memory where everything is shared.

10

class **CAVERN_irbMcastLink_c**

Multicast link management class

A.0.3.0.23 Public Members

void	**unlink** ()	*Unlink this link and hence unlink any keys linked to the multicast channel.*

Multicast link management class. For multicast there is practically nothing to manage except to unlink it when done. Deleting the object will also unlink it.

11

class **CAVERN_irb_c**

Class for the personal IRB returned from CAVERNInit()

A.0.3.0.24 Public Members

	CAVERN_irbId_c	**getSelfIRBId** ()	*Find out what your IRB ID is (in terms of initial IP and port.)*	
11.1	enum	**status_t**	*Status codes returned from various methods.*	151
11.2		**~CAVERN_irb_c** ()	*Deleting the IRB will shut it down*	151
11.3	CAVERN_irbChannel_c*	**createChannel** ()	*Create a channel (either TCP or UDP) over which local and remote keys may share data*	152
11.4	CAVERN_irbMcastChannel_c*	**createMcastChannel** ()	*Create a multicast channel to a multicast group*	152
11.5	CAVERN_irbKey_c*	**define** (CAVERN_irbKeyId_c *newKeyInfo, CAVERN_keyDefAttrib_c *keyDef, status_t *status)	*Define a key on the personal IRB*	153

Class for the personal IRB returned from CAVERNInit(). CAVERNInit() is the only function that generates this object. Each client or server program only has 1 of these objects.

Once you have this IRB object you can call the CAVERN_irb_c::define() member function to define local keys in which data may be stored.

Then you can create a channel (CAVERN_irbChannel_c) between a local and remote IRB and link (CAVERN_link_c) keys over this channel so that any updates to a key will be shared across the network.

11.1

enum **status_t**

Status codes returned from various methods.

A.0.3.0.25 Members

OK	*Operation successful.*
FAILED	*Operation failed.*

11.2

~CAVERN_irb_c ()

Deleting the IRB will shut it down

Deleting the IRB will shut it down. It will also shutdown your application- Nexus won't let me keep the application alive.

11.3

CAVERN_irbChannel_c* **createChannel** ()

Create a channel (either TCP or UDP) over which local and remote keys may share data

Create a channel (either TCP or UDP) over which local and remote keys may share data. After a channel is created it can be opened to connect to a remote IRB. See CAVERN_irbChannel_c::open().

Return Value: a CAVERN_irbChannel_c object or NULL if failed.

11.4

CAVERN_irbMcastChannel_c* **createMcastChannel** ()

Create a multicast channel to a multicast group

Create a multicast channel to a multicast group. After a channel is created it can be opened to connect to the mcast group by using CAVERN_irbMcastChannel_c::open().

Return Value: a CAVERN_irbMcastChannel_c object or NULL if failed.

11.5

CAVERN_irbKey_c* **define** (CAVERN_irbKeyId_c *newKeyInfo, CAVERN_keyDefAttrib_c *keyDef, status_t *status)

Define a key on the personal IRB

Define a key on the personal IRB.

Return Value: A handle to the database. Delete the handle when you're done referencing it. It will help free up system resources.

Parameters: `keyDef` — if set to NULL then it assumes default settings. Currently this is not used so please set it to NULL.

`status` — returns either: OK or FAILED.

12

class **CAVERN_keyDefAttrib_c**

Class describing the properties of a key definition

Class describing the properties of a key definition. This is used in calling: CAVERN_irb_c::define().

Currently there are no defineable attributes. Attributes for setting permissions will be added in time.

13

class **CAVERN_qosAttrib_c**

Class to define network Quality of Service parameters

A.0.3.0.26 Public Members

13.1	enum	**CAVERN_qosGetRule_t**	*Strategies for acquiring QOS*	156
	void	**setQosGetRule** (CAVERN_qosGetRule_t)	*Set QoS rule.*	
	void	**setAcceptableLatency** (float)	*Set acceptable latency in milliseconds.*	
	void	**setAcceptableJitter** (float)	*Set acceptable jitter in milliseconds.*	
	void	**setBandwidth** (int)	*Set bandwidth in bits per second.*	
	CAVERN_qosGetRule_t	**getQosGetRule** ()	*Accessor for QoS rule.*	
	float	**getAcceptableLatency** ()	*Accessor for latency.*	
	float	**getAcceptableJitter** ()	*Accessor for jitter.*	
	int	**getBandwidth** ()	*Accessor for bandwidth (in bits per second).*	

Class to define network Quality of Service parameters. QoS attributes allow you to negotiate how much bandwidth, latency, and jitter is acceptable over your subscription to a remote IRB.

13.1

enum **CAVERN_qosGetRule_t**

Strategies for acquiring QOS

A.0.3.0.27 Members

QOS_GET_EXACT	*Get exact QoS desired or fail.*
QOS_NEGOTIATE	*Ask for QoS but negotiate for best that can be gotten.*

See Also: setQosGetRule, getQosGetRule

14

class **CAVERNplus_callSequencer_c**

Class for enforcing serial processing in CAVERNsoft

A.0.3.0.28 Public Members

14.1	void	**permitCall** (int delay=3)	*Choose a point where an external callback is allowable*	159
	void	**enterCall** ()	*Call this before doing work inside the callback function.*	
	void	**exitCall** ()	*Call this when you are about to exit from the callback function.*	
		CAVERNplus_callSequencer_c ()	*Constructor.*	
		˜CAVERNplus_callSequencer_c ()	*Destructor.*	

Class for enforcing serial processing in CAVERNsoft.

This class is a convenience for people who are trying to retrofit previously developed applications with CAVERNsoft. Often times programs are not written with concurrent processing in mind. CAVERNsoft makes extensive use of concurrent programming and so problems may arise when attempting to retrofit the non-concurrent application with CAVERNsoft. For complex programs this is particularly difficult because there could be many inter-acting variables.

One simple solution to this problem is to force CAVERNsoft to work sequentially too (it's not efficient but for some applications it may be the only course of action.)

The way to do this is to assign a location in your program where you think CAVERNsoft callbacks are safe to occur. Using the CAVERNplus_callSequencer_c class you can declare that point in the program by using the permitCall() member function.

Then from within the callback you can call the enterCall() and exitCall() member functions. Call the enterCall() before doing any work in your callback and do an exitCall() before exiting your callback. Remember to perform the exitCall() function or else your callback will block preventing CAVERNsoft from launching any future calls to this function.

An example of a main() function of a program might be:

```
CAVERNplus\_callSequencer\_c *seq;

main() {

blah....

blah....

// Do this after CAVERNsoft has been initialized.
seq = new CAVERNplus\_callSequencer\_c;

while(1) {
do stuff...

seq->permitCall();
}
}
```

The callback might look like this:

```
callback() {
seq->enterCall();

blah....

seq->exitCall();
}
```

This concept is similar to monitors however unlike monitors the call sequencer mechanism does not explicitly guarantee mutual exclusion for every variable that is accessed within the callback.

14.1

void **permitCall** (int delay=3)

Choose a point where an external callback is allowable

Parameters: `delay` — - the amount of time to wait for a callback to be detected.

15

class **CAVERNplus_condition_c**

Class for thread condition variables

A.0.3.0.29 Public Members

15.1	int	**wait** (CAVERNplus_mutex_c *mutex)	*Wait on a condition to be signalled* ...	161
15.2	int	**signal** ()	*Signal that a condition has arisen*	162
15.3	int	**broadcastSignal** ()	*Signal that a condition has arisen*	162
	nexus_cond_t*	**getNexusCondition** ()	*Return nexus' condition variable.*	

Class for thread condition variables. Condition variables are used in conjunction with mutexes to provide a way for threads to wait until a condition occurs.

An example of waiting on a signal is:

```
// Lock your mutex that is protecting someState.
myMutex->lock();

// Watch for your desired state to occur.
while(someState != reached) {

// Wait for a signal.
myCondition->wait(myMutex);

.... got the condition and the lock so now continue ....

}

myMutex->unlock();
```

An example of sending the signal is:

```
// Lock your mutex that is protecting someState.
myMutex->lock();

// Signal that the state has been reached.
if (someState == reached) myCondition->signal();

// Unlock your mutex so that the waiting thread can continue.
myMutex->unlock();
```

15.1

int **wait** (CAVERNplus_mutex_c *mutex)

Wait on a condition to be signalled

Wait on a condition to be signalled. This function first releases the mutex and then waits on the condition. When the condition arises (ie it has been signaled) the mutex is reaquired, and the function returns.

Return Value:	0 if function successfully completes else non-zero

15.2

int **signal** ()

Signal that a condition has arisen

Signal that a condition has arisen. This wakes up one thread that is suspended on this condition. If no threads are suspended this call has no effect.

Return Value: 0 if function successfully completes else non-zero

15.3

int **broadcastSignal** ()

Signal that a condition has arisen

Signal that a condition has arisen. This wakes up ALL threads that are suspended on this condition. If no threads are suspended this call has no effect.

Return Value: 0 if function successfully completes else non-zero

16

class **CAVERNplus_datapack_c**

Data packing class

A.0.3.0.30 Public Members

	CAVERNplus_datapack_c (char *buff, int sz)	*User must specify a pre-allocated data buffer and size of the buffer.*
void	**setBuffer** (char *buff, int sz)	*User can switch data buffers. But again you need to specify the size.*
char*	**getBuffer** ()	*Retrieve the data buffer.*
int	**getBufferMaxSize** ()	*Retrieve the original full size of the data buffer.*
int	**getBufferFilledSize** ()	*Retrieve the size of the data buffer that has been packed with data so far.*
void	**restart** ()	*Reset the packing position to the beginning of the buffer.*
int	**packFloat** (float val)	*Floats*
int	**packLong** (long val)	*Long*
int	**packInt** (int val)	*Int*
int	**packDouble** (double val)	*Double*
int	**packChar** (char val)	*Char*
int	**pack** (char *buf, int sz)	

Appendix A (Continued)

		*Raw char**
float	**unpackFloat** ()	*Float*
long	**unpackLong** ()	*Long*
int	**unpackInt** ()	*Int*
double	**unpackDouble** ()	*Double*
char	**unpackChar** ()	*Char*
int	**unpack** (char *buf, int sz)	*Char**

Data packing class. It is basically a glorified memcpy(). The idea is that you create an object to help you pack data for transmission over networks.

First you create a CAVERNplus_datapack_c object. Then assign it a pre-allocated memory buffer. Then using the various pack*() member functions, you can pack integers, characters, floats etc into the buffer. When done you can simply assign the buffer to whatever call needs the buffer.

Similarly if you receive a buffer of data from the network, you can unpack its constituent components using the unpack*() member functions. Note: this class does not save the format of your packing or unpacking it simply lays out your data in the buffer you provide it. You must pack things and unpack things in the same order in order for them to make sense.

Author: : Jason Leigh

Version: : 3/7/97

17

class **CAVERNplus_mutex_c**

Mutual exclusion class

A.0.3.0.31 Public Members

	CAVERNplus_mutex_c ()	*Construct for a CAVERN mutual exclusion object.*
void	**lock** ()	*Lock the mutex object.*
void	**unlock** ()	*Unlock mutex object.*
void	**setMutexDebug** (CAVERNplus_mutex_c::mutexDebug_t stat)	*Turn mutex debug messages on or off.*
void	**setMutexDebugMesg** (char *msg)	*Set the debug message to print whenever lock and unlock is performed.*
	~CAVERNplus_mutex_c ()	*Destructor for a CAVERN mutual exclusion object.*
nexus_mutex_t*	**getNexusMutex** ()	*Return the nexus handle to the mutex variable.*

Mutual exclusion class.

This class encapsulates mutual exclusion in a C++ object.

18

class **CAVERNplus_thread_c**

Thread class

A.0.3.0.32 Public Members

Thread class. Note: the CAVERNplus_thread-related classes are intended to be simplified versions of the nexus/pthread interfaces so that beginners can quickly create threaded programs. For greater control over threads use the nexus thread API which is very similar to the pthreads API.

18.1

int **create** (void * (*threaded_func)(void *), void *arg)

Create a thread

Return Value: 0 if thread successful, else non-zero.

Parameters: `threaded_func` — is your function that will be called in a separate thread. The function needs to be of the form: void * threaded_func(void *arg);

`arg` — is the argument to pass to the threaded function.

19

void **cvrnPrintf** (char *fmt, ...)

A thread-safe printf

A thread-safe printf. Use this rather than the standard printf to print text to the TTY.
example 1:
cvrnPrintf("Hello
prints something like:
Hello 123 32.4

Parameters:

`fmt` — format string that you normally would have specified for printf.

`arguments` — for the format string.

Class Graph

Appendix A (Continued)

Appendix A (Continued)

Appendix B

KEYTOOL: THE CAVERNSOFT PERSISTENT HEAP APPLICATION PROGRAMMER'S INTERFACE

Contents

1

class **keyToolKey_c**

Key tool key class

B.0.3.0.33 Public Members

1.1	enum	**status_t**	*Status*	175
	md5Key_c	**getKeyId** ()	*Get the key id of the current key object.*	
1.2	char*	**getData** ()	*Get pointer to data buffer in key*	176
	int	**getDataSize** ()	*Return size of data.*	
	int	**getRealSize** ()	*Return size of internal buffer.*	
	status_t	**alloc** (int size)	*Allocate memory in key (previous contents MAY be deleted).*	
	status_t	**trash** ()	*Delete the data in a key.* *Allocate memory while maintaining contents.* *status_t resize(int size);*	
	status_t	**commit** ()	*Commit this key (automatially timestamps. Does not affect aux time stamp.)*	
	status_t	**setMeta** (char *data, int msize)	*Set meta field*	
	char*	**getMeta** ()	*Get meta data. Returns ptr to the buffer.*	
	int	**getMetaSize** ()	*Return size of meta data.*	

void	**stampTime** ()	*Force stamp the timestamp with the current time. Ie without doing a commit.*
void	**stampAuxTime** ()	*Stamp the aux time stamp.*
void	**setTimeStamp** (double theTime)	*Set the time stamp to a particular time.*
void	**setAuxTimeStamp** (double theTime)	*Set the aux time stamp to a particular time.*
double	**getTimeStamp** ()	*Get the time stamp.*
double	**getAuxTimeStamp** ()	*Get the aux time stamp.*

Key tool key class. Objects of this type are created by the Key Tool Manager.

See Also: keyToolManager_c (→2, *page 177*)

1.1

enum **status_t**

Status

B.0.3.0.34 Members

FAILED	*Failed.*
OK	*Ok.*

1.2

char* **getData** ()

Get pointer to data buffer in key

Get pointer to data buffer in key. Need more elaborate call in future.

2

class **keyToolManager_c**

Key Tool Manager class

B.0.3.0.35 Public Members

2.1	enum	**status_t**	*Status of getKey() call.*	178
2.2	keyToolKey_c*	**getKey** (char *path, char *name, status_t *retStatus)	*Get key*	179
2.3	keyToolKey_c*	**getKey** (md5Key_c keyId, status_t *retStatus)	*Get key*	179
	int	**existsKey** (char *path, char* name)	*Check to see if a key exists at all.*	
	status_t	**swapContents** (keyToolKey_c *key1, keyToolKey_c *key2)	*Swap contents of keys (ptr switch).*	
	status_t	**removeKey** (keyToolKey_c *key)	*Remove a key from this manager.*	
	status_t	**removeKeyPermanently** (keyToolKey_c *key)	*Remove a key from this manager and persistent store.*	
	status_t	**commit** ()	*Commit the entire database.*	
	void	**showProfile** ()	*Show performance profile.*	

Key Tool Manager class. This is a quick keytool simulator hack. It is a first try. The spec will no doubt change over time. So the documentation describes the functionality of the existing classes, which may or may not change in the final spec.

To begin you need to create a key tool manager which basically sets up a directory in which is can store files to hold data for the keys (Ptool will no doubt have its own way of doing this). Currently every key gets 1 data file and 1 meta data file. The data file only contains the contents of the key. The meta data file contains the meta data: size of data, timestamp, comment.

Using the keytool manager you can get a keytool key object (a wrapper) which gives you access to the internals of the keys- ie the actual data, the time stamp, etc. The keytool manager is essentially a cache between physical memory and secondary storage.

Note, the difference between a time stamp and an aux (auxiliary) time stamp is that the former is always done on a commit. The latter is not. The aux time stamp is used to allow user defined timestamping. e.g. the aux time stamp can be used to record the timestamp of the original remote data source, whereas the regular timestamp can be used to record the time stamp of the local copy of the data.

2.1

enum **status_t**

Status of getKey() call.

B.0.3.0.36 Members

FAILED	*Failed.*
KEY_ALREADY_IN_MANAGER	*Key already in manager.*
KEY_FOUND_IN_FILE	*Key read in from file.*
KEY_COMPLETELY_NEW	*Key completely new- ie not in manager nor file.*
OK	*Ok.*

2.2

keyToolKey_c* **getKey** (char *path, char *name, status_t *retStatus)

Get key

Get key. If key does not exist already then a new key is generated in the manager. A wrapper (keyToolKey_c) is returned. This key does not get stored permanently until you do a commit.

If key already exists in the database but not in the manager then load it from the database to the manager. Return a wrapper (keyToolKey_c) to the key.

If key already exists in the manager then simply return a wrapper (keyToolKey_c) to the key.

2.3

keyToolKey_c* **getKey** (md5Key_c keyId, status_t *retStatus)

Get key

Get key. This only returns a keyToolKey_c wrapper if the key has been previously loaded into the manager via the other overloading of the getKey call. If it hasn't then this call returns NULL.

Class Graph

CITED LITERATURE

1. Leigh, J., Johnson, A. E., Vasilakis, C. A., and DeFanti, T. A.: Multi-perspective collaborative design in persistent networked virtual environments. In Proceedings of IEEE Virtual Reality Annual International Symposium '96, pages 253–260, April 1996.

2. Leigh, J., Johnson, A., and DeFanti, T. A.: CALVIN: an immersimedia design environment utilizing heterogeneous perspectives. In Proceedings of IEEE International Conference on Multimedia Computing and Systems '96, pages 20–23, June 1996.

3. Leigh, J. and Johnson, A. E.: Supporting transcontinental collaborative work in persistent virtual environments. IEEE Computer Graphics and Applications, pages 47–51, 1996.

4. Lehner, V. D. and DeFanti, T. A.: Distributed virtual reality: Supporting remote collaboration in vehicle design. IEEE Computer Graphics and Applications, in press, 1997.

5. Cruz-Neira, C., Sandin, D. J., and DeFanti, T. A.: Surround-screen projection-based virtual reality: The design and implementation of the CAVE. In Computer Graphics (SIGGRAPH '93 Proceedings), ed. J. T. Kajiya, volume 27, pages 135–142, August 1993.

6. Locke, J.: An Introduction to the Internet Networking Environment and SIMNET/DIS. Master's thesis, Naval Postgraduate School, August 1995. http://www-npsnet.cs.nps.navy.mil/npsnet/ publications/DISIntro.ps.Z.

7. Macedonia, M. R. and Zyda, M. J.: A taxonomy for networked virtual environments. In Proceedings of the 1995 Workshop on Networked Realities, Oct 1995.

8. Macedonia, M. R., Brutzman, D. P., and Zyda, M. J.: NPSNET: A multi-player 3D virtual environment over the internet. In Proceedings of the 1995 Symposium on Interactive 3D Graphics, pages 93–94. ACM, ACM, 1995.

9. Roussos, M., Johnson, A., Leigh, J., Vasilakis, C., and Moher, T. G.: Constructing collaborative stories within virtual learning landscapes. In Proceedings of the European Conference on A.I. in Education, pages 129–135, Sept 1996.

10. Roussos, M., Johnson, A., Leigh, J., Barnes, C. R., Vasilakis, C. A., and Moher, T. G.: The nice project: Narrative, immersive, constructionist/collaborative environments for learning in virtual reality. In ED-MEDIA/ED-TELECOM 97: World Conferences on Educational Multimedia and Hypermedia and on Educational Telecommunications, 1997.

11. Roy, T. and Cruz-Neira, C.: Cosmic worm in the CAVE: Steering a high-performance computing application from a virtual environment. Presence, 4(2):121–129, 1995.

12. Freitag, L., Diachin, D., Heath, D., Herzog, J., and Plassmann, P.: Remote engineering using cave-to-cave communications. Virtual Environments and Distributed Computing at Supercomputing'95: GII Testbed and HPC Challenge Applications on the I-Way, page 41, 1995.

13. Ware, C. and Balakrishnan, R.: Reaching for objects in VR displays: Lag and frame rate. ACM Transactions on Computer-Human Interaction, 1(4):334–356, December 1994.

14. Park, K. S.: Effects of network characteristics and information sharing on human performance in COVE. Master's thesis, Electronic Visualization Laboratory, University of Illinois at Chicago, 1997.

15. Fish, R. S., Kraut, R. E., and Chalfonte, B. L.: The videowindowsystem in informal communication. In Proceedings of Computer Supported Cooperative Work, pages 1–11, 1990.

16. Tang, J. C. and Isaacs, E.: Why do users like video? In Computer Supported Cooperative Work, pages 163–196. CSCW, 1993.

17. Fish, R.: Bellcore cross-industry working team workshop XIWT. personal correspondence, 1996.

18. Argyle, M. and Cook, M.: Gaze and Mutual Gaze. Cambridge University Press, 1976.

19. McGrath, J.: Groups: Interaction and Performance. Englewood Cliffs, NJL Prentice-Hall, 1984.

20. Short, J., Williams, E., and Christie, B.: The Social Psychology of Telecommunications. Wiley and Sons, 1976.

21. Olson, J. and Olson, G. M.: What mix of video and audio is useful for small groups doing remote real-time design work. In Proceedings of SIGCHI'95, pages 362–368. ACM, ACM Press, 1995.

22. Lin, J. C. and Paul, S.: RMTP: A reliable multicast transport protocol. In Proceedings of IEEE INFOCOM'96, pages 1414–1424, 1996.

23. Mandeville, J., Furness, J., and Kawahata, T.: Greenspace: Creating a distributed virtual environment for global applications. In Proceedings of IEEE Networked Virtual Reality Workshop. IEEE, 1995.

24. Shaw, C. and Green, M.: The MR toolkit peers package and environment. In Proceedings of the Virtual Reality Annual International Symposium. VRAIS'93. IEEE Computer, 1993.

25. Carlsson, C. and Hagsand, O.: DIVE - a multi-user virtual reality system. In Proceedings of the IEEE Virtual Reality Annual International Symposium, 1993.

26. Wang, Q., Green, M., and Shaw, C.: EM - an environment manager for building networked virtual environments. In Proceedings of the Virtual Reality Annual International Symposium. VRAIS'95, pages 11–18. IEEE Computer, IEEE, 1995.

27. Barrus, J. W., Waters, R. C., and Anderson, D. B.: Locales and beacons: Efficient and precise support for large multi-user virtual environments. In Proceedings of the Virtual Reality Annual International Symposium. VRAIS'96, pages 204–213. IEEE Computer Society, IEEE, March 1996.

28. Funkhouser, T. A.: Network topologies for scalable multi-user virtual environments. In Proceedings of the Virtual Reality Annual International Symposium. VRAIS'96, pages 222–228. IEEE Computer Society, IEEE, March 1996.

29. Zhang, L., Deering, S., Estrin, D., Shenker, S., and Zappala, D.: RSVP: A new resource ReSerVation Protocol. IEEE Network, September 1993.

30. Protic, J., Tomasevic, M., and Milutinovic, V.: Distributed Shared Memory: concepts and systems. IEEE Computer Society, 1997.

31. Thiebaux, M.: The Virtual Director. Master's thesis, Electronic Visualization Laboratory, University of Illinois at Chicago, 1997.

32. Foster, I., Kesselman, C., and Tuecke, S.: The Nexus approach to integrating multithreading and communication. Journal of Parallel and Distributed Computing, (37):70–82, 1996.

33. Schmidt, D. C., Harrison, T., and Al-Shaer, E.: Object-oriented components for high-speed network programming. In Proceedings of USENIX Conference on Object-Oriented Technologies, Monterey, CA, Jun 1995.

34. Orfali, R., Harkey, D., and Edwards, J.: The Essential Distributed Objects Survival Guide, chapter 3, page 63. John Wiley & Sons, 1996.

35. TASC Inc.: Report DO0007-96005C-1 : Platform proto-federation: Lessons learned document. Technical report, Prepared for: U.S. Army Simulation, Training and Instrumentation Command (STRICOM), under Contract N61339-95-D-0006, October 1996.

36. Stevens, R. W.: UNIX Network Programming, chapter 6, page 270. Prentice Hall, 1 edition, 1990.

37. Grossman, R. L., Hanley, D., and Qin, X.: PTool: A light weight persistent object manager. In Proceedings of SIGMOD'95, page 488. ACM, 1995.

38. Grossman, R. and Qin, X.: Ptool: A software tool for working with persistent data. Technical Report 93-5, Laboratory for Advanced Computing, University of Illinois at Chicago, 1993.

39. Grossman, R., Lifka, D., and Qin, X.: An object manager utilizing hierarchical storage. In Twelth Symposium on Mass Storage Systems. IEEE, IEEE Press, 1993.

40. Baden, A. and Grossman, R.: Database computing in high energy physics. In Computing in High-Energy Physics 1991, eds. Y. Watase and F. Abe, pages 59–66, Tokyo, 1991. Universal Academy Press, Inc.

41. Cruz-Neira, C., Sandin, D. J., DeFanti, T. A., Kenyon, R. V., and Hart, J. C.: The cave automatic virtual environment. Communications of the ACM, 35(2):64–72, June 1992.

42. Shaw, C., Liang, J., Green, M., and Sun, Y.: The Decoupled Simulation Model for Virtual Reality Systems. In Proceedings of CHI '92, pages 321–328. ACM, May 1992.

43. Funkhouser, T. A., Sequin, C. H., and Teller, S. J.: Management of large amounts of data in interactive building walkthroughs. In Computer Graphics (1992 Symposium on Interactive 3D Graphics), ed. D. Zeltzer, volume 25, pages 11–20, March 1992.

44. Zyda, M. J., Pratt, D. R., Monahan, J. G., and Wilson, K. P.: NPSNET: Constructing a 3D virtual world. In Proceedings of the 1992 Symposium on Interative 3D Graphics, pages 147–156, 1992.

45. Singh, G., Serra, L., Ping, W., and Ng, H.: BrickNet: A software toolkit for network-based virtual worlds. Presence: Teleoperators and Virtual Environments, 3(1):19–34, 1994.

46. Gaver, W., Sellen, A., Heath, C., and Luff, P.: One is not enough: Multiple views in a media space. In Proceedings of INTERCHI'93, New York, Apr 1993. ACM, ACM.

47. Morningstar, C. and Farmer, F. R.: Cyberspace: first steps, chapter The Lessons of Lucasfilm's Habitat, pages 273–302. MIT Press, 1991.

48. Blanchard, C. and Burgess, S.: Reality Built for Two. In Symposium on Interactive 3D Graphics. ACM SIGGRAPH 90, 1990.

49. Grimsdale, C.: dVS: Distributed virtual environment system. Division Ltd, 1992.

50. Loefler, C.: Networked virtual reality. In ATR Workshop on Virtual Space Teleconferencing, pages 108–119, 1993.

51. Codella, C., Reza, J., Koved, L., and Lewis, J. B.: A toolkit for developing multi-user, distributed virtual environments. In Proceedings of IEEE Virtual Reality Annual International Symposium '93, pages 401–407, 1993.

52. Bricken, W. and Coco, G.: The VEOS project. Technical report, Human Interface Technology Laboratory, University of Washington, 1993.

53. Shu, L. and Flowers, W.: Groupware experiences in three-dimensional computer aided design. In Proceedings of the ACM Conference on Computer Supported Cooperative Work, pages 179–186. ACM, ACM Press, 1992.

54. Stansfield, S.: An application of shared virtual reality to situational training. In Proceedings of the Virtual Reality Annual International Symposium. VRAIS'95, pages 156–161. IEEE, IEEE Computer Society Press, 1995.

55. Macedonia, M. R., Zyda, M. J., Pratt, D. R., Barham, P. T., and Zeswitz, S.: NPSNET: A network software architecture for large-scale virtual environments. Presence, 3(4):265–287, 1994.

56. Benford, S., Fahlen, J., Greenhalgh, L. E., and Snowdon, D.: User embodiment in collaborative virtual environments. In Proceedings of SIGCHI'95, pages 242–249, New York, NY, 1995. SIGCHI, ACM.

57. Lehner, V.: Caterpillar collaborative vehicle design: `http://www.ncsa.uiuc.edu/VEG/DVR`. Technical report, National Center for Supercomputing Applications, 1996.

58. Greenhalgh, C.: MASSIVE95 `http://www.crg.cs.nott.ac.uk/ cmg/massive95.html`, 1995.

59. NCSA: Habanero. http://www.ncsa.uiuc.edu/ SDG/Software/Habanero/index.html, 1996.

60. Roussos, M., Johnson, A. E., Leigh, J., Vasilakis, C. A., and Moher, T. G.: NICE: Narrative immersive constructionist environments (http://www.ice.eecs.uic.edu/~nice), 1996.

VITA

Jason Leigh is a PhD candidate in his final moments of his degree specializing in Teleimmersion where he can combine his interests in human-factors, interactive computer graphics, databases, and art. He has been working at the Electronic Visualization Laboratory at the University of Illinois at Chicago for 5 years. In that time he has worked on a wide range of projects that include:

Several VR visualization applications for Computational Neuroscientists at CALTECH as part of the federal Human Brain Project which were exhibited at: SIGGRAPH'92, Neuroscience'93, and SIGGRAPH'94.

Developing a multimedia presentation for Al Gore that was shown during the signing of the Telecommunications Act of 96.

Worked in collaboration with artist Christina Vasilakis on an art show that allows visitors to tour a virtual house of the future. This project was Jason's first introduction into the world of Teleimmersion.

This preliminary work led to the construction of CALVIN a collaborative architectural design system which has the unique property of allowing participants to interact with each other simultaneously using heterogeneous perspectives. This work was presented at VRAIS'95.

Currently he is working on three projects:

He is assisting General Motors in applying collaborative technologies to their VisualEyes VR vehicle design system.

He is working as a member of the NICE project to design a VR collaborative educational environment for kids.

Finally he is developing a new collaborative VR networking architecture (called CAVERN-soft) that integrates networking and databases in a manner that is optimized for Teleimmersion.