Providing Efficient Networking for Distributed Virtual Reality Using CoRgi

Thesis

Submitted in partial fulfillment of the

requirements for the Degree of

Honours in Computer Science

of Rhodes University

by

Bryan Kilian

November 1998

Abstract

This thesis presents an implementation of a number of network protocols and designs, for integration into the CoRgi Virtual Environment component toolkit. First, a history of networking in virtual reality is presented, followed by a discussion of the different types of data requiring transport over a distributed system of this type. In the next chapters, different distributed networking architectures are presented, with case studies of three widely differing Virtual Environment systems and their solutions to the inherent problems in large scale networking of Virtual Environments.
Performance issues and challenges related to networking in Virtual Environments are then presented and discussed.

The CoRgi networking design is then presented, with a discussion of the reasons for the design decisions made in the networking design. Finally, some examples of programming using BSD sockets are given followed by the implementation specifics of the networking system and the results of tests performed to measure efficiency of the different components.

Table of Contents

1. Introduction

Virtual Reality, where a person immerses himself in a computer-generated world, is fast becoming the current catchword in Computer Science. It is used for an array of diverse projects. One of the most common is scientific visualisation, where the scientist finds himself in the center of the sun, or the center of an atom, with equal ease. Another use is in medicine; a surgeon can use a VR generated image of the patient’s insides, without the problems of blood and objects obscuring his view.
The military uses it for training purposes, saving themselves millions in equipment costs. [Brutzman 1997]
All these uses however, are not completely realistic in that there is no interaction with other humans involved. Bringing in the interaction component generates a whole host of new problems. We now have to address the mechanisms of connecting more than one person to a virtual world. This could be done with all the people in the same room, but that defeats the object. Ideally we want to use this technology for connecting people across the globe to one another, and allow them to interact normally.
The infrastructure for doing this exists already, in the ever faster global internetwork known as the Internet. Using this network, people can connect to a shared resource (such as a virtual environment) and all interact with it and each other simultaneously. If the Internet had infinite bandwidth, this would be ideal, and we would be able to use simple connections between the clients, with no worries about slowdowns and data content.
The Internet is, however, a scarce resource, and bandwidth has to be carefully managed to achieve good results. The burden for this falls to the networking system built into the server and/or clients in the system. Virtual Reality requires a lot of bandwidth to be realistic, and for activities such as teleconferencing, requires even higher bandwidth for functions like video streams and full audio.

There are a number of systems trying to address the problem of managing the limited resource of the network in applications such as this. I will discuss some of them and their strengths and shortcomings later in this document.
Our goal is to create an efficient networking subsystem for use in distributed Virtual Reality, and more specifically, for integration into the CoRgi pilot project.

2. Background

It' s clear that there are a number of obstacles to be overcome in achieving this goal. The fact that we want people to be able to access if from their homes means that we have to be able to run over relatively limited-bandwidth links, such as 28.8k modems. The fact that we want to run over the Internet means that we have to tolerate a certain amount of latency is the delivery of update information. Finally, the fact that people are running on different computer systems with different hardware and different software means that we must design the system for portability.
When deciding on a way to approach implementing the networking server support for CoRgi I looked at a number of other systems, and chose a method that I thought would best benefit the way CoRgi works. I couldn’t just apply another established method, since CoRgi’s component based system did not lend itself to the way other networking implementations are built.

2.1 Networking in VR

Since the advent of the internet and global networking, distributed virtual reality has grown from a fledgling idea, implemented in text based multiplayer games, to full graphical immersive worlds, the main limitations being the bandwidth available. As bandwidth increases, we find more uses for it, sending higher detailed streams, and creating vaster and better virtual worlds.

2.1.1 Networking Basics

The Internet is based around a family of protocols called TCP/IP. The IP (Internet Protocol) part is the lowest level, it handles addressing and routing. IP packets can be sent over Ethernet, fiber optics, telephone lines, radio links, tin cans and string, or any other medium that can move bits.

Above IP are two other protocols: TCP and UDP. TCP, the Transmission Control Protocol, provides a connection-oriented, reliable byte stream form of communication. Applications can use TCP to open a logical connection to another host on the Internet, send data down that connection, and receive data back. TCP uses IP to actually move the packets to other hosts.

TCP is the protocol on which most Internet applications run. The Web uses HTTP (Hypertext Transport Protocol) which runs over TCP; when you click on a link, a connection is opened to another host using TCP, and the data transfer is carried out through this connection. Similar techniques are used for Finger, SMTP (electronic mail) and NNTP (USENET news). In all cases, the actual networking is handled transparently by TCP.

UDP, the User Datagram Protocol, is quite different from TCP. It provides a connectionless, unreliable way of sending datagrams (packets) from one host to another. The word "unreliable" in this context doesn' t mean that UDP is error-prone, it just means that there' s no guarantee that a specific packet will arrive. However, if a packet does arrive, it will arrive intact. NFS (the Network File System), NTP (Network Time Protocol) and several others use UDP.

There are some important advantages in using UDP instead of TCP. The problem with TCP lies in its strengths. TCP connections have a lot of associated overhead, because the TCP protocol is sending information about packet contents, flow control information, CRC information and other data which TCP uses to ensure good service. UDP packets are relatively "lightweight". Although they get lost occasionally, there' ll never be any congestion; they won' t clog things up, they' ll just be discarded. Also, unlike TCP, UDP packets can be broadcast on subnets (since there' s no "logical connection" involved).

However, because UDP is "unreliable", it requires the use of "stateless" protocols; in other words, each message must be complete and self-contained, and make no assumptions about previous messages having been received. NFS is an example of a stateless protocol.

2.1.2 History, Multiplayer games to VR.

This is a brief history of the events leading up to today's Internet and its use in Distributed VR applications. In looking at VR as an escape from reality, I noted that it had grown from the ideas of early multiplayer games, where people would become whatever they chose and battle computer created monsters, or just socialise in a fantastic surroundings. The early VR applications had the spirit of what we have today, even though they did not have the technology to implement it.

In October 1963, Thomas Marill, Daniel Edwards, and Wallace Feurzeig published "DATA-DIAL: Two-Way Communication with Computers from Ordinary Dial Telephones" in *Communications of the ACM.* (Vol 6, Number 10). The host computer was a DEC PDP-1. This was the first application that allowed users to communicate with computers remotely without using special expensive equipment. BBN also registered the patent on the modem on June 17, 1963, and subsequently developed the foundations of modern computer networks. [Burka 1995]

In 1976, Don Woods at the Stanford AI labs released a multiplayer version of the game ADVENT by Will Crowther, on the PDP-10. This game later became known as Adventure, but the OS on the PDP-10 only allowed 6 character filenames.*

Roy Trubshaw, a student at Essex University, wrote the very first MUD (Multi-User Dungeon) in BCPL on a DEC-10 in spring 1979. It was originally just a series of interconnected areas where people could gather and chat. It used the EPSS (Experimental Packet Switching System) link Essex had to ARPANet at the time. [Walsh et al 1995]

It developed from there into numerous different types of multiplayer games, all exclusively text based, due to the limited bandwidth. These virtual worlds relied on

* ADVENT was responsible for a number of terms gamers and hackers now take for granted, such as "You are in a twisty maze of passages, all alike" and the all too common root password "xyzzy".

text descriptions and keyboard input, coupled with the user's imagination. They have remained popular to this day, being used for scientific conferences as well as games. These games were completely server based, not having to do any graphics rendering, they used a simple telnet client on the user's side as a front end. The networking was very simple, needing a single text connection per player.

As the bandwidth availability increased, it became possible to consider putting graphics onto the front end for VR games. This led to games like CrossFire, and PennMUX, which then had to find ways to send graphical data efficiently across slow links. They used specialised clients, which interpreted the data and rendered it simplistically on a display. This was still very far from the VR we see on movies such as Lawnmower Man, but had more realism and required less imagination on the user's part.

When the Web and VRML were invented, it took the idea of Distributed VR up another level, now being able to create entire 3D worlds, which were rendered on the client side. This is still mostly graphical, with very little audio and other feedback. It also lacks realism due to simplified objects to minimise bandwidth.

With current 3D hardware, we can now start making realistic VR worlds, but this requires high bandwidth, and a well designed networking infrastructure.

2.1.3 Networking Support for VR.

Virtual Reality applications are specialised in the type of data they use, and therefore in the types of data they would send over the network. As the level of realism grows, so does the network bandwidth required. Different types of data also have different requirements. For instance, you would require a sound stream to coincide with the action happening, even if the sound quality dropped slightly to accomplish this. Similarly, a stream of position data would not have to be so precise, if we used only every third value, our motion might be slightly jerkier, but it would still be usable. From this, we can classify the network data that we would send into four main types.

[Macedonia et al 1995] and [Brutzman 1997] contend that these four types of network data are

- Lightweight Interactions – State data, entity interactions, in other words, data that can be encoded in a single packet.
- Network Pointers – Also lightweight data, which tells a client where to find another network resource.
- Heavy Weight Objects – Large datasets, which need a reliable connection, and contain entity definitions or other data that is small enough to fit into the client's memory.
- Real time streams – Large video, audio or other streams, which need to be delivered with as little latency as possible.

[Leigh et al 1997] took a slightly different view, and used classifications of

- Small Event Data – Data that needed priority delivery, and could fit in one packet. The equivalent in previous classifications would be Lightweight Interactions and Network Pointers.
- Medium Atomic Data – The same as Heavy Weight Objects mentioned previously.
- Large Segmented Data – Similar to the Real time Streams, but not restricted to only streaming data, any data that would need to be abstracted down into manageable portions before use fit here.

I decided to use a classification similar to [Leigh et al 1997] and classify my data types as State Changes, Objects, Real Time Streams, and Non Critical Streams.

2.1.3.1 State Changes

State changes encompass any change to the VR world that affects the objects in it, for instance, it could be the attributes on the object itself changing, either personal ones (damage taken, size) or world related ones (position, direction). It could also be interactions between objects, collisions, stray rockets…

These could be handled in two ways, either as a peer-to-peer system, where each player/object notifies the other object itself, or done in a server-controlled manner, where the server checks for all interactions and notifies the objects accordingly.

There are positive and negative aspects to both of these options. In the peer-to-peer case, it makes interactions faster, and not affecting the whole world, but only the objects associated with the interaction. It also means that the objects have to notify the world of the result of the interaction, and have to be more complex in that they have to know how to deal with other objects themselves.

In the server case, the objects themselves are simplified, as the server modifies attributes depending on results of interactions, but this places a higher load on the server, with less happening client side.

In network terms, the peer-to-peer system would require a far more complex network layout, with the objects connecting to the server as well as to every other object. The complexity rises quickly with the number of objects and while it prevents a bottleneck of having all traffic going through the server, the amount of traffic itself is greater.

The server layout is far simpler, looking like a star based topology, with each client interacting through the server. This introduces the bottleneck of the server, but reduces network traffic if the interactions are taking place on the server and only attributes being changed on the objects.

2.1.3.2 Objects

VR systems in general use an object-oriented view, as it translates to the "real world" quite well. An object would normally consist of a number of attributes, defining the object, and it's interactions with the rest of the world. It might have graphical or sound attributes associated with it as well as behaviours.

A client would have to have a way to render the VR world for a person to view. This could be done in a number of ways, the client could either have a representation of the world around it in object terms, or that could be kept on the server only, and the server send out graphical information, either as triangles, or as raw graphical data.

The least network intensive solution would be for the client to have a smaller representation of the world around it, the server would then pass either updated or new objects to the client, or modify the attributes on all the objects. This could also lead to a client that controls its own interactions and passes changes back to the server.

In both cases Objects would have to be transferred along the network, since objects all have similar form, this could be easily optimised to reduce bandwidth even more.

2.1.3.3 Real Time Streams

For best realism, the VR world would have to have audio, video and other high bandwidth streams associated with it. Background noises, or even speaking audio streams would make the experience far more enjoyable and useful. These streams require different handling to the previous two, they cannot easily be encoded, since they do not have a structure that promotes this.

Audio and video streams have requirements where it is possible to decrease quality (resolution or colour depth) and it can still be acceptable as long as there is no skipping or stalling. This means that if you encode the data correctly, it could easily be done automatically by the networking system by just dropping packets that are deemed unnecessary.

Finding a balance between CPU usage and network usage is quite difficult where high bandwidth data like audio or video are concerned, there is hardware available that would do encoding, but if you are considering software, then it becomes more difficult.

2.1.3.4 Non Critical Streams

A number of data types are either purely informational, or a client gets sent far more than it needs to do an accurate calculation. The client can discard the extra data items, but they still use the network and take up bandwidth. These streams can use a lossy protocol like UDP because losing a packet every so often makes no impact on the rest of the system

The types of data that have these qualities are, for instance, position sensor data. Position sensors typically send thousands of data items a second, the client only needs say sixty a second for an application that is rendering at 60 frames a second. Even 25 packets a second would be acceptable.

Other data fitting into this category would be if a server were filling in details for the client. It could do so as long as there was bandwidth available, when the bandwidth is restricted, then less detail gets through, and the client would render a blockier, less realistic image, but still at respectable speed. [Brutzman 1997]

2.2 Distributed Designs

2.2.1 Server Centered Approach (Figure 1)

This approach uses a centralised server, where the clients all connect to that server. It is the simplest architecture to maintain, since the clients do not need complex networking implementations. All the transactions and communication are handled by the server.

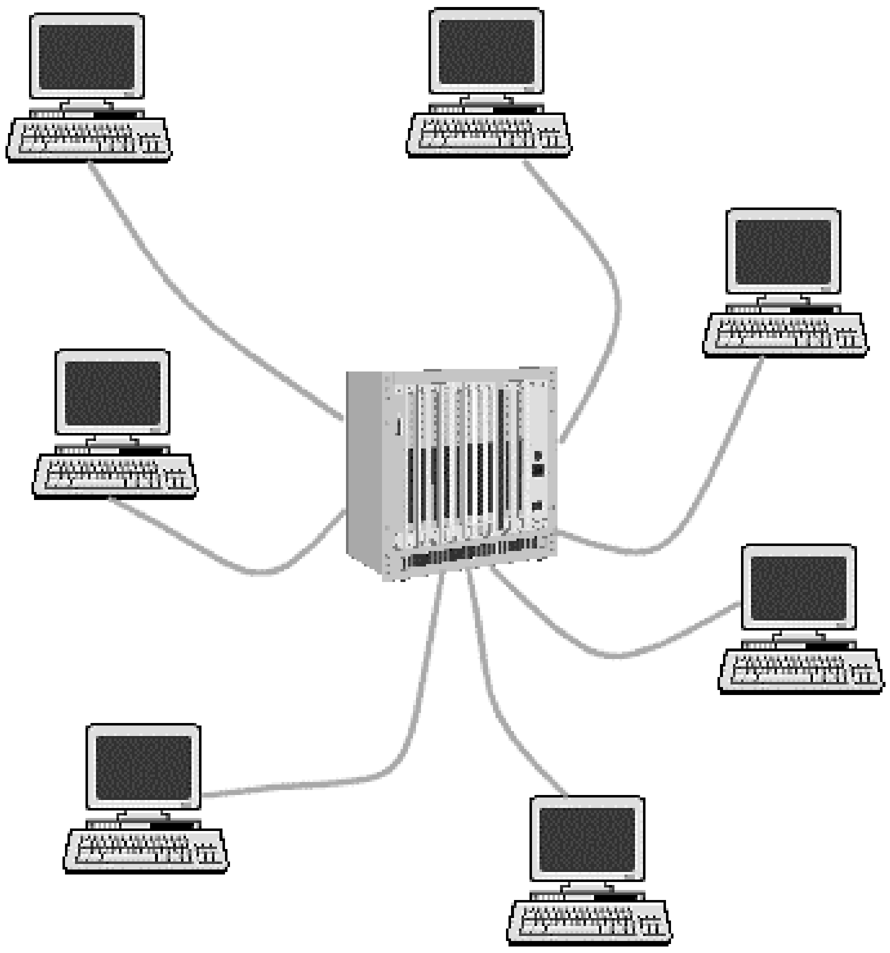


Figure 1 – Server based Architecture

The type of client-server communication will determine the type of data flowing on the network, and the complexity of the server. There are numerous different topologies for data sharing with this approach, with the easiest being the MUD style centralised database and "dumb" dient approach. Other topologies can include the clients having a copy of the database, which gets updated by the server, or the clients having a portion of the database, relating to their current environment and position. The major disadvantage to this network topology is that the server becomes a bottleneck. In all cases, the bandwidth the server has to have is N times the client bandwidth, with N clients. The exact amount of bandwidth depends on the type of environment, and the amount of data passing between the client and the server.

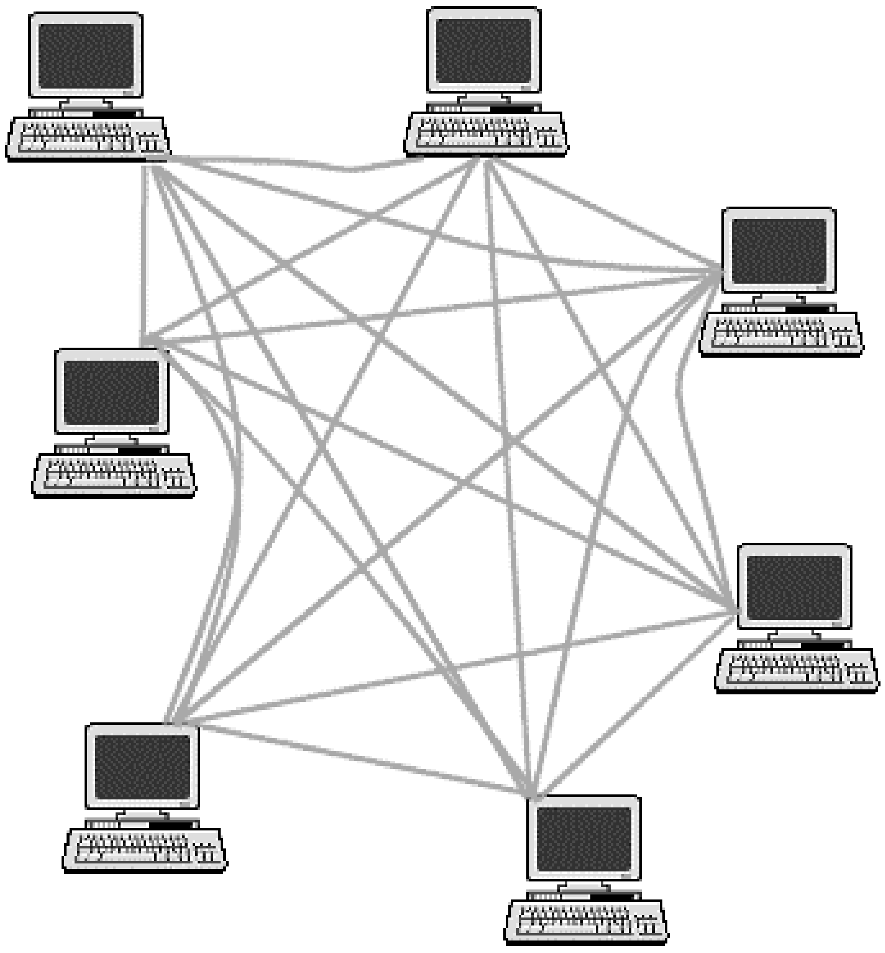


Figure 2 –Peer-to-peer network configuration

2.2.2 Peer-to-peer Approach (Figure 2)

This approach decentralises the control, with all the clients connected directly to all the other clients. This allows fast efficient transport of data between clients, but can be a problem if there is a large amount of data that has to go to all clients. The networking implementations on the clients in this case can be very complex.

When large world updates occur, a client can easily exceed its bandwidth if there are a large number of clients on the network. This approach also requires the clients to be able to store the entire world. In effect, every client runs the entire VR world, and synchronises with all the other clients connected to it.

It does, however, simplify matters where multimedia data is concerned. The client can send the audio or video feed directly to the receiver, without having anything filtered or slowed down by a server. Using Multicast technology, the bandwidth requirements for multimedia data can also be greatly decreased.

The main disadvantage to this approach is a lack of network control from a centralised position. It would be difficult to do traffic shaping and control on a VR environment running in this manner. The client complexity is also far larger than the single server case, requiring advanced network management routines to be built into every client.

2.2.3 Network Broadcast (Figure 3)

The simplest approach, and the one that some multi-player computer game systems use, is to simply have each host broadcast the location of each entity that it maintains. These broadcasts are received by every host participating, and are used to update their local copy of the database. In this way it is similar to the Peer-to-peer system, but differs in that it uses the network broadcast address for its information flow, and so does not need to keep track of other connected clients.

The first networked version of the computer game "Doom" worked in a broadcast mode; each participant constantly broadcast the current state of his or her avatar. Most multi-player arcade games work the same way.

This approach works acceptably on small, dedicated networks; however, there are a number of problems with it. The most important problem with broadcasting is that every machine on the subnet must receive and process every update packet; this includes machines that aren' t participating in the simulation! That' s not a problem on a dedicated LAN, but networks that are being used for other things can' t have huge amounts of broadcast traffic on them; this is why so many companies and universities adopted a "no network DOOM"[*] policy.

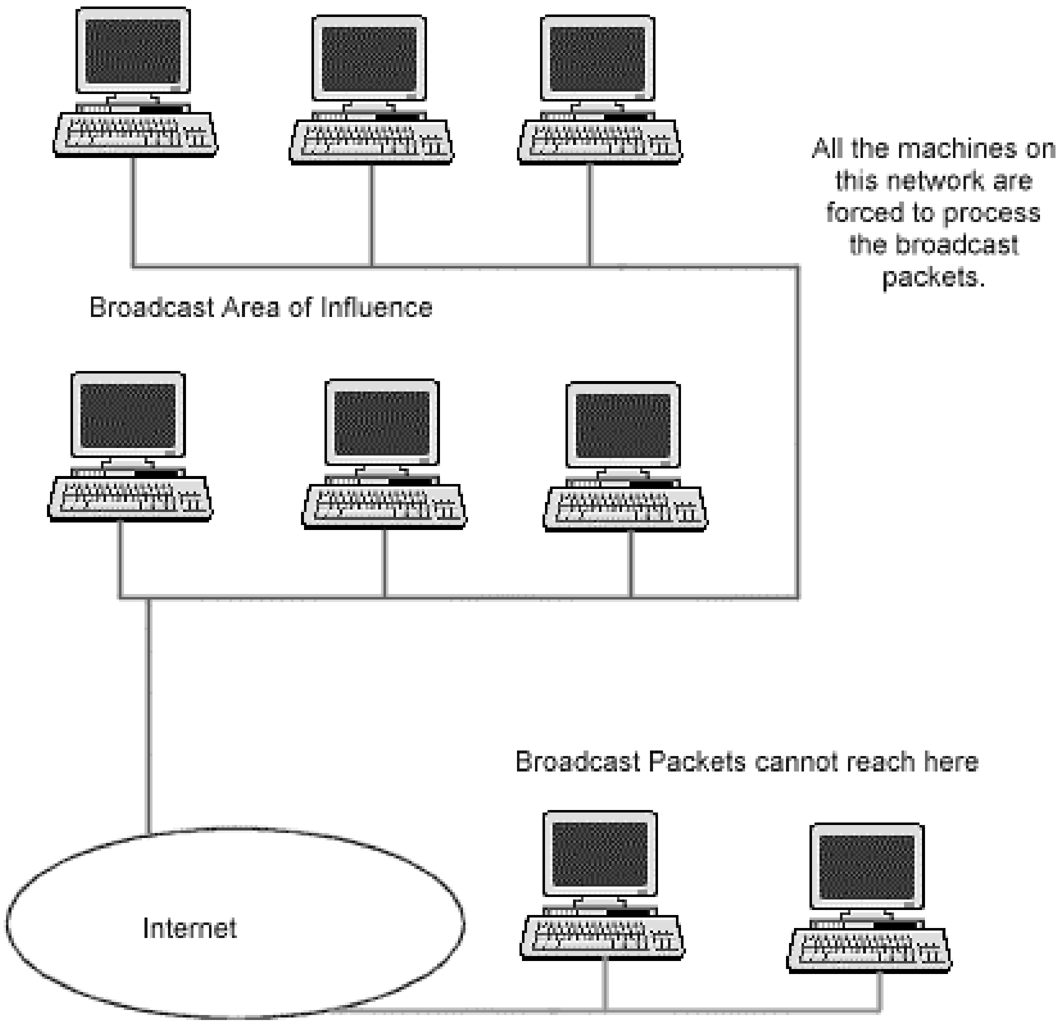

All the machines on this network are forced to process the broadcast packets.
Broadcast Area of Influence
Broadcast Packets cannot reach here
Internet

Figure 3 – Network using broadcast packets

[*] DOOM, incidentally, switched to point-to-point messages for subsequent releases in order to be friendlier to non-dedicated networks

Another drawback is that broadcast traffic does not, as a rule, cross routers, and so a workaround would have to be installed for people in other subnets to be able to participate.

This approach does not scale up, as the number of clients increase, the limits of the physical network become apparent.

A modified version of this method uses the IP Multicast address instead of the broadcast address (Figure 4).

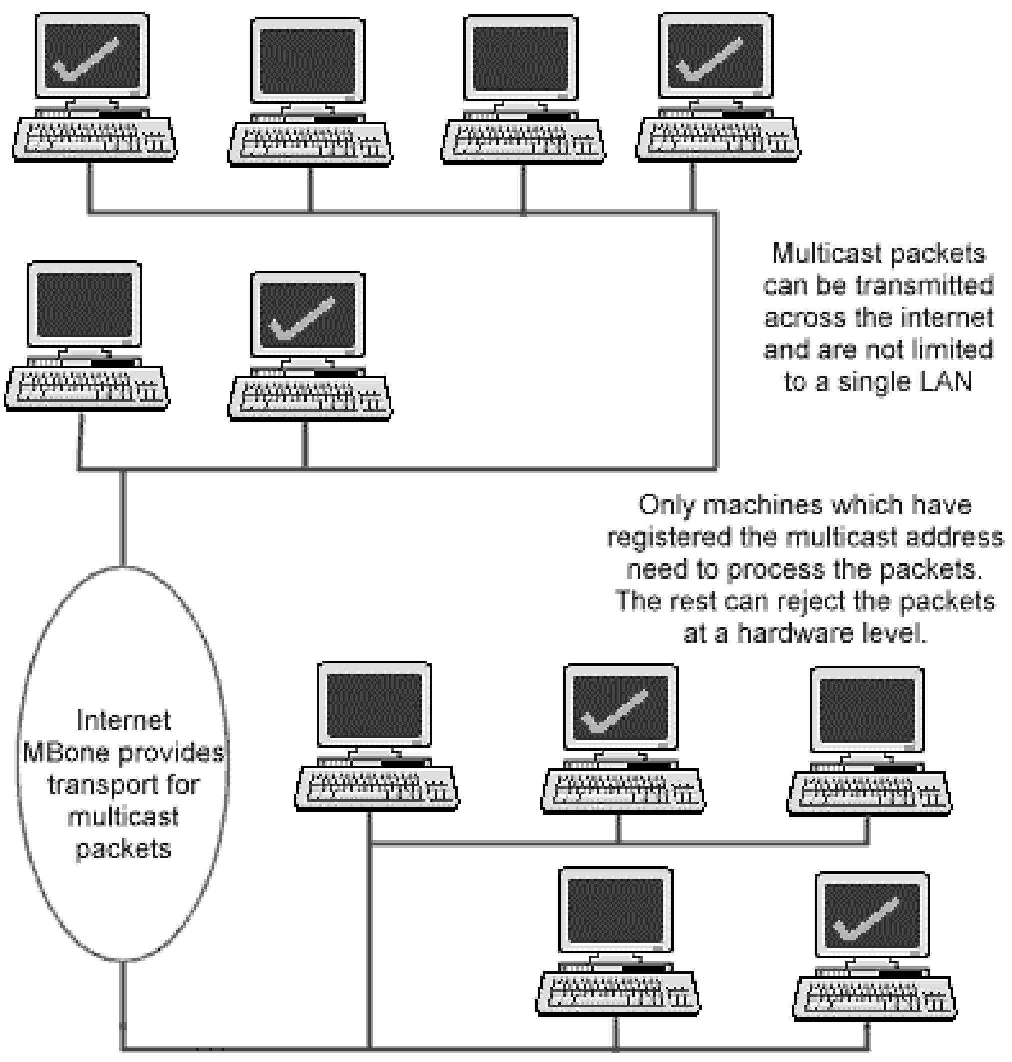

Multicast packets can be transmitted across the internet and are not limited to a single LAN
Only machines which have registered the multicast address need to process the packets. The rest can reject the packets at a hardware level.
Internet MBone provides transport for multicast packets

Figure 4 – Multicast Distributed configuration

2.2.4 Hybrid (Figure 5)

This is the approach taken by [Mclean 1997] and it provides the highest configurability and scalability. It uses a server to maintain the Environment, and replicated databases on the client, to minimise the network traffic needed for updating. The Server provides control information and major world updates. It also is where clients announce themselves to join the system, the server can then update the client with a copy of the replicated database and the client can the join the simulation. Each client can also communicate with every other client as necessary. The methods to do this could be peer-to-peer, broadcast or, more likely, UDP multicast. The server can be used to resolve issues where the data of two clients differ.

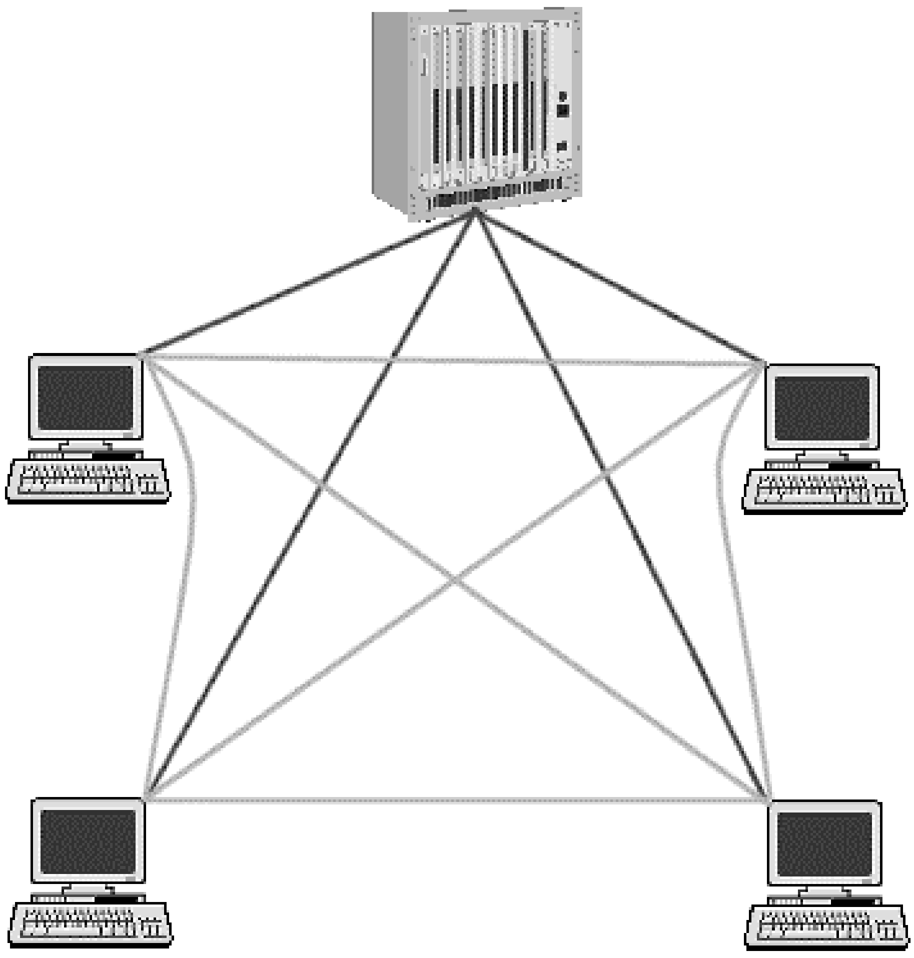


Figure 5 – Hybrid network configuration.

[Mclean 1997] suggests that the sever can be used for tasks such as seamlessly connecting worlds, reliability, resolving contention issues, security, and event distribution. This last task is an interesting one; the server will store the regions in the world and the multicast addresses associated with them. When a client passes out of the influence of a region, the server will provide it with the address for the new region it is entering. This corresponds to the "Area of Interest Manager" in the DIS scenario discussed in section 2.2.5.1 [Macedonia et al 1995]

2.2.5 Case Studies

While researching the best system to use for implementing my network structure, I came across a number of other Distributed VR projects. There are too many to include them all here as case studies, but I wanted to include a range of projects, not just the biggest and best. I decided to include NPSNet, Astro-VR and CAVERN, which together cover most of the issues I have noticed with Distributed VR networking. Some projects, like the Caterpillar Inc. Distributed Virtual Reality Project [NCSA 1998], while being of interest, did not cover anything that the three projects I have featured here did not deal with.

When looking at other VR networking designs, my concerns were mostly how I could extract their successes and apply them to the CoRgi system, while avoiding obvious problems associated with the system. I had the advantage of not having to build around an already existing established networking system, and so had some leeway in my design.

2.2.5.1 NPSNet's DIS

The largest, best-known and most successful standard for distributed VR has been DIS -- the Distributed Interactive Simulation protocol. It was developed at the Naval Postgraduate School in Monterey, California. Not surprisingly, DIS, like its predecessor SIMNET, is designed for a very specific application domain: military simulations. [Roehl 1995]

It addresses the problem of compatibility by defining a standard message format for interchanging information between hosts. This standard format is called a PDU, for Protocol Data Unit. There are many different types of PDU, but they are specialised for military type data like requests for resupply of munitions. They transfer data about the entities in the world using a format named the "Entity State PDU"

To reduce the bandwidth requirements, DIS uses dead reckoning. This sends a velocity vector instead of a positional point for the entity, thus the rest of the hosts in the system can extrapolate the behaviour and position of the other entities in the system without requiring constant positional updates from them.

As [Mclean 1997] points out, this method reduces the number of updates required significantly, and therefore the network bandwidth requirements are also reduced. The approach works impressively for the domain DIS covers, since every entity in the environment is a vehicle or projectile, it is inherently suited to the dead reckoning ideology.

DIS uses UDP broadcast packets to send its PDUs to other hosts. Each ESPDU has the complete state of the entity, and they're re-broadcast every few seconds (or more often, if the entity determines that more frequent updates are needed). It also uses cells of Multicast addresses to filter updates.

[Macedonia et al 1995] proposed an extension to the NPSNET system that would allow for far greater scalability, the use of "regions of influence". They modeled the VR environment on behaviours they had witnessed in actual battle simulations. They noticed that a particular vehicle would only be affected by vehicles in the immediate vicinity, and in a simulation, a third of the vehicles never moved very far. Using this, they constructed a model for partitioning the environment into equal sized areas, with each area having it's own multicast address. An "Area of Influence Manager" managed areas and the participants in them. (See Figure 6)

Unfortunately, for a generalised VR Environment, DIS would not be suitable, the packet formats are far too specialised, and dead reckoning does not work as well for unpredictable or complex behaviours.

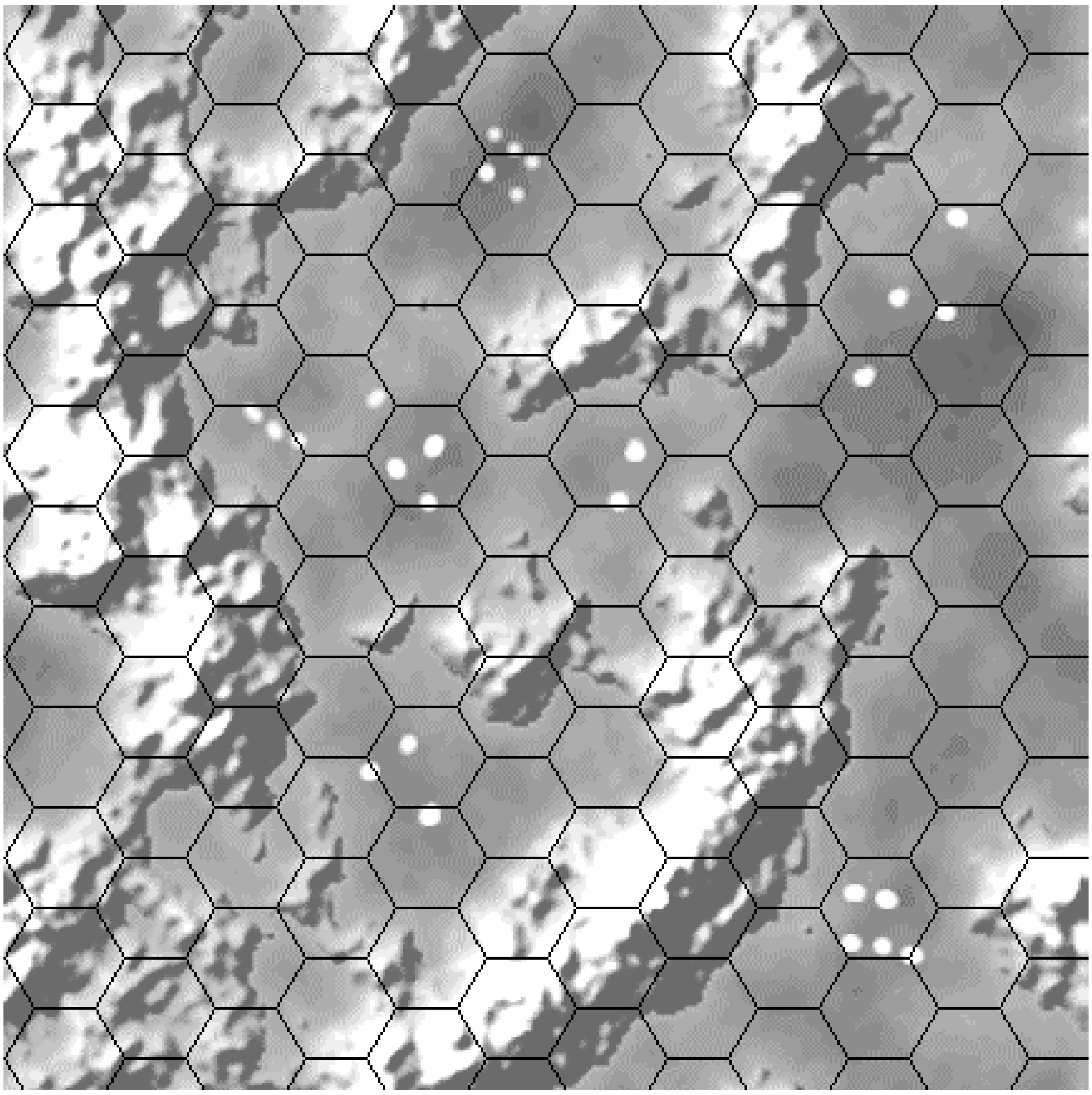


Figure 6 – Using hexagonal regions to partition a VR environment

DIS provides us with a number of useful ideas:

- Using standard message formats.
- Having a fully decentralised model with no central server. The DIS model assumes all clients have a full copy of the database.
- Use of Multicasting to filter updates to clients in the immediate vicinity.

2.2.5.2 CAVERN

"CAVERN, the CAVE Research Network, is an alliance of industrial and research institutions equipped with CAVEs, Immersadesks, and high performance computing resources, interconnected by high-speed networks to support collaboration in design, training, education, scientific visualization, and computational steering, in virtual reality" [Johnson et al 1998]

The CAVE project started out as a scientific visualisation tool. It now connects three continents with high-speed data lines for experimentation in highly immersive environments.

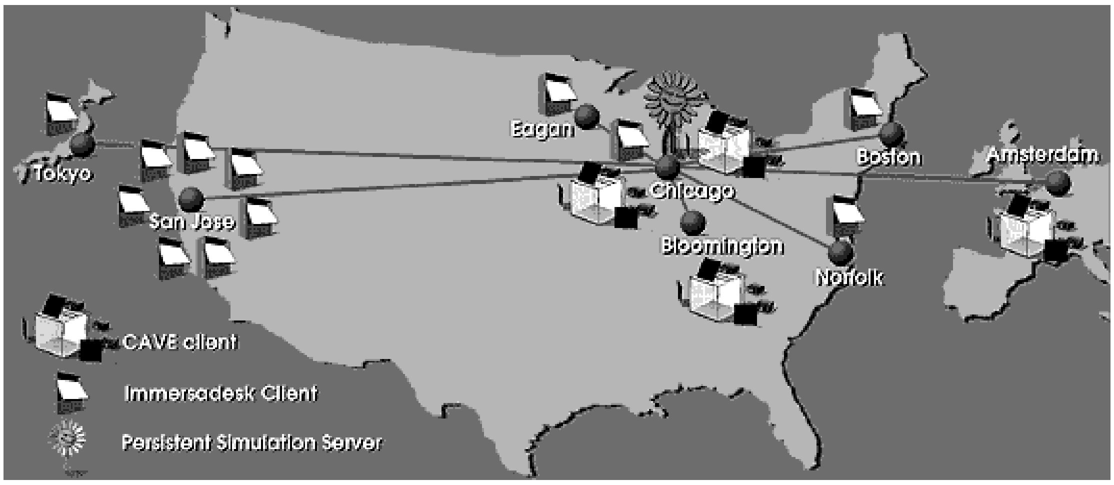


Figure 7 – One CAVERN Configuration

The CAVERN system uses a system called CAVERNsoft as it's controlling application, CAVERNsoft consists of a nucleus called the Information Resource Broker (IRB).

The IRB is a small, highly configurable nucleus that can be imbedded into every distributed software system that is expected to participate. The IRB is spawned as a thread, with both networking and database capabilities, so that it can serve both client and server needs.

IRB-based applications communicate with each other by establishing single or multiple communications channels. Each channel is individually customized to meet the specific needs of the VR data being transmitted. The application specifies whether it wants reliable TCP, unreliable UDP or unreliable multicast and the desired bandwidth, latency, and jitter.

After communications channels have been established, each application may create an arena of data that can be linked to one another so as to emulate a distributed shared memory (DSM) segment between the connected applications. Using the same channel or different channels, many arenas can be created. Each of these can be used to store different forms of data (avatar tracking, 3D models, etc.) [Johnson et al 1998]

In addition to the many automatic networking capabilities provided by the IRB it still supports 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 helps manage basic socket-level interfaces by providing automatic mechanisms for accepting new connections. [Leigh et al 1997]

The CAVERN project has a number of concepts we can apply.

- The networking code is a separate component from the VR application.
- It allows the system to use a number of protocols.
- It tries to maintain a good Quality of Service

2.2.5.3 The Astro-VR Project

Another attempt at adding on to established codebases, the Astro-VR project was undertaken by the Palo Alto Research Center as a means of creating a virtual environment for the international astronomy community. I have listed it here because it uses a completely different approach to the DIS system discussed above.

Astro-VR is, quite simply, VR extensions to a standard MUD server (namely the Object Oriented LambdaMOO server). Thus it inherits all the networking options applicable to standard MUDs, and has been extended to support multicasting of video and audio data. The aims of the Astro-VR system are:

Real-time multi-user communication. The standard MUD server architecture, as well as audio extensions achieve this.

A self-contained electronic mail and bulletin board system. Once again, built into most standard MUD libraries, and easily modified by using the MUD’s built in language.

Shared user-supplied links to online astronomical images. This is achieved by extending the MUD architecture to include an "Image" object, which can be viewed by anyone. The dedicated client will fetch the image from the place it is stored (URL) and display it in a window for the astronomer.

An editor/viewer for short presentations of text and images. This is once again achieved by utilising the built in MOO language, and designing virtual conference rooms, that have special commands associated with them. The conferences are even archived for later perusal.

Collaborative access to standard programs used by astronomers. The clients provide this functionality by interfacing with the server. Users can "virtually" collect around a whiteboard and discuss their ideas while using the applications that they would normally use. This is important in that it does not force a user to use a proprietary application, it allows the system to be used with their normal apps.

Networking wise, this approach uses the Server centered idea, with a centralised server controlling all access and controlling the client interaction. It works well with few users and low bandwidth data, such as the standard MUD text. It starts falling short when a number of people start collecting and using the high bandwidth possibilities of the system, such as video, audio, and shared application use.

This system is useful in that it allows full reprogramming of the environment from inside, with a full-featured object oriented language. A user can create an object that opens windows on another user's machine, and interact with it.

This system works well for it's application, which is to provide a collaborative VR environment for a small group of people. They also acknowledge the problems of high bandwidth usage, and propose using UDP multicasting for the high bandwidth streaming data.

Here again, there are a number of useful concepts that struck me.

- The design is simple.
- The clients do not have to do very much world processing, thus allowing a more heterogeneous environment.

- While being simple, it wasn't very extensible, and had a severe shortcoming; the server had to be powerful enough to do all the processing for the entire world.

2.3 Performance Issues

2.3.1 Challenges

In a perfect world, with unlimited bandwidth and CPU power, implementing VR Environments would not be problematic. You could just put together a simple networked system, and not have to worry about slowdowns and bandwidth. Today's world is far from perfect, however, and we have to make sure our Environment meets a number of challenges and conquers them before it can be successful.
[Cowcroft 1998] characterises some of the challenges as

- Capacity - bits per second (otherwise known as bandwidth) Which may differ between a sender and a set of recipients
- Errors/Loss - packet loss may be due to noise on the line, transient outage, or temporary congestion.
- Availability - networks can partition. Some senders or recipients may crash, or reboot at different times during a multi-party session.
- Delay - delays over large networks can be significant.

[Roehl 1995] lists the challenges as being ones of compatibility, limited bandwidth, and latency and [Macedonia 1995] lists them as Latency, Reliability and Bandwidth. I looked at all the classes listed and split them into three groups, compatibility, resources, and latency.
I will give a breakdown of challenges and performance issues using these three categories.

2.3.1.1 Compatibility

Compatibility affects a number of performance issues, and touches every aspect of networking in today's environment. From issues as low level as the order in which a particular architecture stores its bytes in memory, to the operating systems used and speed of Internet connections. The computing environment available today for distributed processing is a heterogeneous one. Earlier VR implementations such as early implementations of NPSNet required certain hardware and high bandwidth lines to be in place before the system would work.

Today, however, with the advent of fairly cheap, moderately powerful PC's and workstations, the VR systems are moving to gain a foothold on this market, and increase the userbase and subsequently the acceptance of the techniques by mainstream users. The problems introduced by this, however, need to be overcome before the current system can perform efficiently. The VR systems must now take into account that the bandwidth different clients have may be different. It cannot assume they have equally powerful CPU's and also it cannot assume that the machines even store their data in a similar manner.

The clients want these problems to be resolved transparently, with no intervention on their part. This requires the building of more "intelligent" VR systems that can gauge their client's performance and act on it.

A good example of this is using feedback optimisation for graphics performance as tested by [Fred]. This technique can also be applied to the networking subsystem, where the network can monitor itself and try correct bandwidth and performance issues as they occur.

2.3.1.2 Limited Resources

A problem related to the one discussed above, where different clients have different bandwidth lines, is the fact that no matter how wide a pipe the machine is on, it will not be enough for your application. The eventual possible size of your distributed environment will mainly be limited by two things, the number of packets your clients can process, and the amount of bandwidth available to you.

Although the telco companies are trying valiantly to remedy the situation, the bandwidth available on the Internet is not infinite. This is fine for a few clients, or low quality multimedia, but for the best results, you want to have as many clients as possible, and good quality audio/video.

[Macedonia et al 1995] proposed a solution to extend the current NPSNet/DIS system to support far more users by using multicast groups. Using multicast enabled them to send one packet which was picked up by numerous machines, while avoiding the trap of broadcasting and forcing every machine on the network to process the packet. They proposed splitting the environment up into hexagonal areas, with each area defining an area of influence. Objects in an area could not affect objects in another area, and this allowed clients to process only the packets for their particular area.

[Macedonia 1995] states that the DIS system would require 375Mbps links to every workstation in a network to support 100 000 participants, which is an unrealistic number to be cost effective still today.

While network resources are one major bottleneck, the second is processor performance. Even using 40 processor SGI Onyx machines, providing realistic physical interactions and collision detection becomes practically impossible when reaching more then 1000 objects.

This problem will not be completely solved by increases in CPU power and bandwidth, it requires a well designed network structure for the distributed Environment to minimise the amount of information that a single client has to process.

2.3.1.3 Latency

Some sites will have a very low latency (lag) and some may have a high latency factor. The system must take this into account and try to deliver the best possible service to both sides. We do not want a slow computer or a computer on a slow link to hold up the rest of the system.

3. Network Extensions to CoRgi

3.1 The CoRgi Approach

CoRgi was not designed from the start with a particular network architecture model in mind. This makes it quite difficult to apply some of the lessons learned from other VR systems.

The strength behind CoRgi is its modular structure. It is possible by coding simple "components" to plug in any functionality you need. There are a number of different networking modules for CoRgi already, providing most base level networking functionality.

The UDPChannel provides the base UDP functionality in a pseudo point-to-point manner. There are no components to deal with broadcast or multicast traffic.

3.2 Channels And NetworkComponents

Channels are a base class that provides us with a generalised interface to network data streams. A channel encapsulates the network handshaking and low level parts of networking, presenting a higher level interface to the user, who considers a channel to be just another stream of data that needs some initial setting up.

A Channel is considered by the system to be a single point of input and output. The channel itself may have multiple connections, but these would not be distinguishable by the rest of the system. If the channel receives data, it is considered to be one source, and data sent to the channel is similar in that one cannot specify a destination. If the channel has multiple connections, it would normally act as a broadcaster.

Channels are subclassed to provide functionality for different network protocols, to enable the use of another protocol, one would have to create a subclass of channel, and build the network functionality into that. Currently the system supports TCP Channels, and UDP Channels. The UDP channel emulates a connection-oriented interface, but will drop packets if necessary.

A NetworkComponent extends the Channels idea by interfacing it to the standard CoRgi component-based structure. It takes one Channel (generalised) and manages it with a simplified interface that the VR system can use.
Channels can be used to provide the peer-to-peer networking structure used in the complex systems like NPSNET. With a Broadcast and Multicast option, it can be extended to give the same service as peer-to-peer, with far less network usage.

3.3 Network Server Component

Continuing with the design decision to make the system as flexible as possible, while still keeping the usage as simple as possible, I decided to implement a NetworkServer, which would provide the services normally required of the single server multiple client system.
The NetworkServer component provides a standard interface for including a client server model into your application. It accepts connections of any sort, and handles them much like a normal multiclient server. You can broadcast to all connected channels or send data to only one.
Since the server deals with TCP traffic, some streams may be large. The network usage can benefit from the application of a compression algorithm to these streams, reducing the stream size, and therefore the number of packets needing to be sent.
I decided to design the system with two compression types, Zlib, and RLE (Run Length Encoded). Zlib requires some setup data, and for small data groups does not achieve a smaller data size.
When I conducted tests using the Zlib library, the breakeven size for ordered incremental data was 281 bytes at the default compression level.
Using random data, zlib did far worse, which was to be expected, since Huffman encoding is not effective if the all the characters in the stream are equally likely to occur. It did, however, indicate that the rand() function wasn't particularly biased.
The RLE algorithm I used compressed up to 256 characters in a run to 3. And if the token char was found it converted to 2 bytes. This is an extremely simple and fast

algorithm, but only works on datasets that contain a lot of repeating characters (more than 3 at a time).
As a result, I included both algorithms in the server.
A built in lag detector is used to determine if a stream should be compressed.

4. Integration into the VR Environment

4.1 Environment Interface

The Environment interface design was influenced by the need for a standard message format in the CoRgi system, the current system requires the VRActors (clients) to be compiled into the binary. They are designed to call the functions in the VREnvironment directly. I did not want to require that all the current VRActors be recoded to reflect the new system.
The Message design had to be transparent to the Actors, so I decided the best approach would be a design that the actor could call as it would normally do, and the Environment interface would translate between function calls and message types automatically.
The attributes that an Actor can set in the CoRgi system are position, orientation, scale, velocity, acceleration, mass, elasticity, force, type and select. I propose a single packet for all these attributes, an "attribute" packet. Each attribute may have a number of different calls, such as SetAttribute, SetWorldAttribute (use world coordinates), SetAbsoluteAttribute (Use absolute coordinates).
There are other calls that an actor can make to affect the Environment, such as can be seen in Listing 1. These could have their own messages.
Once we have decided on a packet standard for the system, implementing the Environment Interface would require writing an object that inherits from VREnvironment, and therefore can be linked together with a VRActor. This imposter object would then convert the calls to our packet format, and pass them along to an

object impersonating a VRActor on the server side. The results would get passed back in a similar manner.

Having the client wait for a response from the environment enables us to make the network use a "reliable" method, if this includes UDP interactions, reliability can be achieved by having the environment send an acknowledge packet. If the client does not receive an acknowledge packet within a certain time, it sends the original packet again.

This method would introduce obvious slowdowns in the interactions, with the client waiting on network input most of the time. [Macedonia 1995] correctly points out that this would also limit scalability. A possible solution would be implementing dead reckoning in the Environment (as suggested by [Mclean 1997]), and have a separation of packets that require the Environment to respond. Packets falling into the first category could be object attribute changes, which may only be sent once in a session. Packets in the second category (or stateless packets) that do not need a response from the world would be position or velocity packets, which will generally be numerous and easy to correct when packets are lost.

In this way, when the client calls a function that sets a non-stateless attribute, it is blocked until a reply is received from the network. If it tries to change a stateless attribute, the interface returns immediately with the expected return value, and then transmits the packet without waiting for confirmation from the rest of the system.

With the final design of the network server, the Environment interface becomes more interesting. Linked with a network server, or with UDP multicast channels, depending on the application requirements, its job is to propagate information from the network into the Virtual Environment. One design I had in mind would be similar to the DIS system, where there are a defined set of objects, and the client sends a status packet to instantiate itself into the Environment. The Interface would then link that network address to an object in the Environment and translate all further packets to affect this object. This design is easily extended to a point where the database is fully distributed, and UDP multicasted packets manage updates.

This would require a different Environment Interface, one that forwarded the call onto a local Environment object, as well as sending the state packets out onto the network. For a Server model, the system could use unreliable UDP for update and state packets, and a reliable TCP connection to the server to receive VRSink Data in a compressed stream.

4.1.1 Performance Issues

With the Interface sitting between the Environment and the network, it would have to be efficient enough to process and manage all the connections to it, from inside and outside the system.

4.1.2 The Transparent Interface

Creating a system that would not require us to redesign the rest of the CoRgi system was a priority in this case. I decided to take an idea from the Java RMI methodology and design a "transparent" interface, that the clients would not realise that there was a network between them and the Environment. While the method used would not be RMI in the technical sense, the result to the client would be similar. A System that abstracted the network and required little to no change in the current VRActor objects, or the VREnvironment object. The design of the specific message types and packets could be specific to each Environment Interface, depending on the requirements of the system they were designed for. Providing a system that used DIS packets in this manner would be entirely feasible. The design is sufficiently generalised that it could be extended to work with any protocol and VR packet type.

4.1.3 Pitfalls

The pitfalls with this system would be the obvious complexity and the fact that you might have a number of different Interfaces, all designed for a specific purpose. Also, every change in the VREnvironment code would require a corresponding change in all the VR interfaces, to support the new calls.

```
/** Set the shape of an object in the environment. */
void setPhysicalRepresentation(VisualRepresentation *shape, objectID me);

/** retrieve the shape of an object in the environment. */
VisualRepresentation *getPhysicalRepresentation(objectID me);

/** Set some/all of the attributes of an object in the environment. These may be
    relative to the structure parent position. */
void setAttribute(AttributeSet *some, objectID me);

/** Get some/all of the attributes of an object in the environment. Value returned
    must be deleted by the caller after use. Values returned are relative to the
    parent. */
AttributeGet *getAttribute(objectID me, attrflags flags, attrmodes mode);

/** add a new object to the environment. */
objectID createThing();

/** remove an object from the environment. */
void destroyThing(objectID me);

/** set a relationship between two objects in one of the hierarchies. */
int makeRelationship(objectID parent,objectID child, relationship hierarchy);

/** break a relationship between two objects in one of the hierarchies. */
int unmakeRelationship(objectID parent,objectID child, relationship hierarchy);

/** return the object colliding with the given one, assuming the results of
    the last call are passed as the second parameter. Returns INVALIDOBJECTID
    when the whole world has been searched, and this value should be
    passed as the initial value. If a valid collision occurs the distance
    between object centers is returned in distance. */
objectID GetCollision (objectID me, objectID lastcollide, double & distance);
```

Listing 1 – A sample of the VREnvironment.h header file.

5. Implementation

The implementation was carried out on Linux and IRIX, with the zlib compression only being used on Linux, since the IRIX machines we had did not have zlib installed. When I started this project, I only had a hazy idea of network programming. I had written server code for a MUSH (Another type of MUD), but that code was more concerned with the mechanics of the server. The networking code we inherited from an earlier version on the system. While I had a good theoretical background in networking, the specifics of actual socket programming had never been covered in any course. I spent many hours poring over the cryptic Unix manual pages and going through reams of code before I had a good understanding of exactly how to open my first socket. For this reason, I'll include a section here condensed from my experiences, about the actual mechanics of network implementations. This should also give the reader a better idea of the reasons behind some choices made in the CoRgi networking implementation. I will be referring back to the following chapters in my discussion of the CoRgi system.

5.1 Network Protocols

Applications that send data over the network generally use an implementation of the BSD (Berkeley Systems Distribution) sockets network architecture code. A socket is a software representation of the endpoint to a communication channel. Sockets can represent many different types of channels, including reliable communication with a single destination host, unreliable communication with a single destination host, unreliable communication with multiple destination hosts, or even in-memory communication with another application on the local host. Regardless of the type of communication channel, the application sees a common abstraction.
A socket identifies several pieces of information about a communication channel:

- Protocol: How the operating systems exchange application data. The protocol implies a level of reliability by specifying whether the destination operating system will send acknowledgment packets and whether packets should be re-transmitted if no acknowledgment is received.

- Destination host: The destination address, or addresses, for packets sent on this socket. In some cases, the destination address is not stored with the socket, in which case the application must specify one whenever it sends a packet. Some addresses have special characteristics associated with them, like the multicast set of addresses, and the broadcast address.
- Destination ID, or port: This number identifies the appropriate socket on the destination host. For each protocol, each of a host's sockets is assigned a different 16-bit integer by the operating system. By specifying the protocol along with this port number in each packet, the source host ensures that the destination host can deliver the packet to the correct application. Just like with the destination host address, some sockets may not have a destination port specified, in which case it must be specified whenever the application sends a packet.
- Source host: This address identifies which host is sending the data. This information is rarely needed at the source host because there is only one choice in most cases. However, if the host has multiple Internet addresses assigned to it, it can legally specify a different source address in each data packet.
- Local ID, or port: A 16-bit integer that identifies which application is sending data along this socket. By including this port number along with the source host address in the packet, the source host ensures that the destination host will be able to send reply packets back to the sending application.

Most hosts on the Internet today use the Internet Protocol (IP) to communicate with each other. IP is a low-level protocol used by hosts and routers to ensure that the packets travel from the source host to the destination host. IP hides the fact that the transmission path might include phone lines, Local Area Networks, Wide Area Networks, wireless radios, satellite links, or carrier pigeon.

Applications almost never actually use the Internet Protocol directly. Instead, they use one of the protocols that are written on top of IP, like TCP or UDP. These higherlayer protocols provide support for application port numbers, and they also include services such as acknowledgments and retransmission. The next section discusses these protocols and how to program them.

5.1.1 TCP

The Transmission Control Protocol (TCP) is the most common protocol in use in the Internet today. When layered on top of the Internet Protocol (IP), it is more commonly referred to as TCP/IP. TCP/IP provides the application with the illusion of a simple point-to-point connection to an application running on another machine. TCP/IP appears to be reliable, because it automatically transmits acknowledgment packets and retransmits data.

Each endpoint can regard a TCP/IP connection as a bi-directional stream of bytes between the two endpoints. TCP/IP automatically takes care of the networking issues of lost packets, packets arriving out of order and the extraction of the actual data from the packets. The protocol also allows the application to detect when the other end of the connection has disconnected.

The simplicity and ease of use of TCP does come with a drawback, of course. Because of its reliability and ordering code, TCP/IP must transmit more information to describe the data ordering, checksums to detect corruption, and lost packets. The receiving application cannot easily decide to skip parts of the stream if it falls behind. It must receive and accept the entire stream as the other end transmitted it. Consequently, though TCP/IP is useful for a large variety of applications, it is not suitable for applications that do not necessarily need the strict ordering and consistency guarantees that TCP/IP provides.

The first step in using TCP/IP is to obtain a socket. The **socket()** function is used to obtain one as shown in listing 2. To now use that socket to connect to a remote host

```
int sock;  /* Socket descriptor */
struct sockaddr_in serverAddr; /* Address Structure */
/* Allocate a socket
  PF_INET: Use the Internet family of Protocols
  SOCK_STREAM: Provide reliable byte-stream semantics
  0: Use the default protocol (TCP) */
sock = socket(PF_INET, SOCK_STREAM, 0);
/* Clear the address structure */
bzero((char *)&serverAddr, sizeof(serverAddr));
serverAddr.sin_family = PF_INET; /* Use Internet addresses */
serverAddr.sin_addr.s_addr = inet_addr("146.231.29.2");
/* htons() converts a 16-bit short integer into the network byte order so
that other hosts can interpret the integer even if they internally store
integers using a different byte order */
serverAddr.sin_port = htons(2401);
/* Connect to the remote host */
if (connect(sock, (struct sockaddr *)&serverAddr, sizeof(serverAddr)) == -1)
       return;
```

Listing 2 – Creating a BSD Socket

and application it must allocate a **sockaddr_in** (Internet socket address) structure that specifies the destination host address and port. Listing 2 then continues by connecting to a server at host address 146.231.29.2 and port 2401. The **connect()** function call does several different things. First, it picks a free local port and binds that port to the client’s socket. The client usually does not care which port number it gets assigned because other applications do not connect to it. Second, the function attempts to contact the server at the specified address and port. If the connection is accepted, it initializes the connection so that the application can send and receive data over the socket.

For a server to accept an incoming connection, it has to set up a listening socket on the relevant port as in Listing 3. It allocates the socket as before, and then **bind ()**s the socket to the port it wants to listen on. The **bind ()** call may fail if another application has already bound that port. Once the socket is bound to a port, it calls the **listen ()** function to tell the operating system how many connections it will accept to that port, and then the **accept ()** call to actually accept a connection. Handling many connections can either be done by using separate threads for each connection, or by using the **select ()** call.

Once a connection is established, the system can send and receive data from the socket. To send data, you place the data in a buffer and pass the buffer to a **write ()** call. Receiving data is done through the **read ()** call. The catch is that the blocks of data are not atomic, and a **read ()** might not return all the data that was sent with a corresponding **write ()**. It is up to the application to make sure it has received the entire block of data it was waiting for, even though it may have to make multiple **read ()** calls to do so. A simple way to notify the receiving end of the size of a block of data is to make the first byte, or word, depending on the general length of blocks of data you are sending, the length of the data in that block. The receiver can then make sure they keep issuing the **read ()** call until they have received the entire block of data.

When handling multiple sockets, you must remember that the **accept ()** and the **read ()** calls block until a connection is made or data is received. This could cause problems in a system that is required to do things other than just accept socket connections, or receive data from a socket. The system provides a mechanism to deal with such problems in the **select ()** call. This call allows a server to detect activity on multiple sockets from a single call, doing away with the need to otherwise use resource expensive processes, and the associated management problems. The **select ()** call takes a bitmask representing a set of sockets to check, and returns when one or more of them have a connection ready to be accepted, or data waiting to be read. On return, it notifies the application of which sockets can be manipulated without blocking.

```
struct sockaddr_in serverAddr; /* The address and port of the server */
struct sockaddr_in clientAddr;
int acceptSock, sock;

bzero((char *)&serverAddr, sizeof(serverAddr));
serverAddr.sin_family = PF_INET; /* Use Internet addresses */
/* INADDR_ANY says that the operating system may choose to which local IP
address to attach the application. For most machines, which only have one
address, this simply chooses that address. The htonl() function converts a
four-byte integer long integer into the network byte order so that other hosts
can interpret the integer even if they internally store integers using a
different byte order */
serverAddr.sin_addr.s_addr = htonl(INADDR_ANY);
serverAddr.sin_port = htons(2104);
/* Bind the socket to the well-known port */
if (bind(sock, (struct sockaddr *)&serverAddr, sizeof(serverAddr)) == -1) {
       /* Error */
       return;
}

acceptSock = sock;
listen(acceptSock, 4);
while ((sock = accept(acceptSock, (struct sockaddr)&clientAddr,
sizeof(clientAddr))) != -1) {
/* sock represents a connection to a client, clientAddr is the client's host
address and port */
/* ... Process client connection ... */
}
/* Only break out of loop if there is an error */
```

Listing 3 – Accepting a TCP connection

5.1.2 UDP

UDP (User Datagram Protocol) is a lightweight protocol. It differs from TCP in three respects, it has connectionless transmission, best-efforts delivery and packet-based data semantics.

UDP does not establish peer-to-peer connections and therefore the sender and recipient of UDP data do not keep any information about the state of the communication session between the two hosts. Such information was used by TCP to detect packet loss, request retransmissions, and manage the connection. With UDP, therefore, none of these features are available. Instead, UDP simply provides "best-efforts delivery" meaning that it makes no attempt to guarantee that data is delivered reliably or in-order. The data is transmitted with no guarantees that it will reach the destination. The sender must rely on other information (such as responses from the recipient) to determine whether the recipient is still alive, whether the data arrived, and whether the recipient can keep up with the data. Finally, because the endpoints do not maintain any state information about the communication, UDP data is sent and received on a packet-by-packet basis.

These datagrams must not be too big, since if IP must fragment them, some could get lost in transit, forcing the receiver to throw away the entire packet.

At first, it may appear that UDP/IP is too weak to be useful. However, its power lies in its simplicity. Because it does not include the overhead needed to detect reliability and maintain connection-oriented semantics, UDP packets require considerably less processing at the transmitting and receiving hosts.

Because UDP/IP does not maintain the illusion of a data stream, packets can be transmitted immediately instead of waiting in line behind other data in the stream. Similarly, data can be delivered to the application as soon as it arrives at the receiving host instead of waiting in line behind missing data. Finally, many operating systems impose limits on how many simultaneous TCP/IP connections they can support.

Because the operating system does not need to keep UDP connection information for every peer host, UDP/IP is more appropriate for large-scale distributed systems where each host communicates with many destinations simultaneously.

These characteristics reveal why UDP/IP has traditionally been the protocol of choice for large-scale Distributed VR systems. In these systems, data must be sent directly to multiple recipients, and real-time data delivery is important.

One aspect of UDP/IP can make it unsuitable for some environments. When a socket is receiving data on a UDP port, it will receive packets sent to it by any host, whether it is participating in the application or not. This possibility can represent a security problem for some applications that do not robustly distinguish between expected and unexpected packets. For this reason, many network firewall administrators block UDP data from being sent to protected hosts from outside the local network.

As with TCP/IP, the first step in using UDP/IP is to obtain a socket. A UDP socket is obtained using the **socket()** function, except with the **SOCK_DGRAM** parameter to indicate UDP datagram protocol. With this socket, the host can now start sending data to any destination. Note that there is no need to call **connect()** because UDP/IP is connectionless.

The local port number is chosen randomly by the operating system when data is first sent along the socket. However, in most applications that use UDP/IP, it is desirable to bind the UDP socket to a well-known local port, just like a TCP/IP server, by calling the **bind()** function.

The **bind()** call may fail if another local UDP/IP application has already bound to that port.

TCP/IP and UDP/IP port numbers are independent of each other, meaning that a TCP/IP application and a UDP/IP application may simultaneously use the same port numbers on the same host.

To send data using UDP/IP, applications place data into a buffer and provide that buffer to the **sendto()** function. When invoking this function, the application also provides a **sockaddr** structure describing which host IP address and UDP port should

receive the packet. The fourth parameter specifies a set of special-delivery flags. Note that because UDP data is sent in packets (as long as the data is within a maximum size range), the transmitted buffer does not need to explicitly include the packet length.
UDP protocol states that if a packet reaches it's destination, it will arrive intact, therefore if the data fits into one packet, it will always arrive in one cycle.
Receiving UDP data involves a call to the **recvfrom()** function. This function takes a buffer for holding the received data and it fills a provided **sockaddr** structure with information about which host IP address and UDP port transmitted the data.
Finally, when either side has finished sending and receiving UDP data, it can close its socket with the **close()** call.
Of course, because there is no connection, closing a socket simply cleans up local system resources. It is important to remember that the **close()** call only closes the local socket.
Unlike with TCP, other hosts will never find out that an endpoint has closed its UDP socket, except that they will no longer see any data from that host/port.
The applications must have agreed to stop communicating, or they must have some other way of determining that a peer has departed.

```
/* ... */
struct sockaddr_in localAddr, destAddr, srcAddr; // Address structures
int sock;                                    // The socket
// PF_INET: Use the Internet family of Protocols
// SOCK_DGRAM: Provide best-efforts packet semantics
// 0: Use the default protocol (UDP)
sock = socket(PF_INET, SOCK_DGRAM, 0);
if (sock == -1) { // Error
       return;
}
bzero((char *)&localAddr, sizeof(localAddr));  //
bzero((char *)&destAddr, sizeof(destAddr));    //  Clear the Address
bzero((char *)&srcAddr, sizeof(srcAddr));      //  structures

localAddr.sin_family = PF_INET;                // Use Internet addresses
localAddr.sin_addr.s_addr = INADDR_ANY;        // Use any local IP address
localAddr.sin_port = htons(2401);              // Port that others can send to

// Bind the socket to the well-known port
if (bind(sock, (struct sockaddr *)&localAddr, sizeof(localAddr)) == -1) {
       return;                                 // Error
}
/* . . . */
// Send data to the specified destination
destAddr.sin_family = PF_INET;                 // Use Internet addresses
destAddr.sin_addr.s_addr = inet_addr("146.231.29.2");
destAddr.sin_port = htons(2104);               // Destination Port.
if (sendto(sock, buf, strlen(buf), 0, (struct sockaddr *)&destAddr,
       sizeof(destAddr)) != strlen(buf)) {
       return;                                 // Error
}
/* . . . */
// Receive data sent to the UDP port
if (recvfrom(sock, buf, sizeof(buf), 0, (struct sockaddr *)&srcAddr,
       sizeof(srcAddr)) == -1) {
       return;                                 // Error
}
// Sender's address stored in srcAddr structure
```

Listing 4 – Code for UDP communications

5.1.3 UDP Broadcast

With UDP/IP, an application can direct a packet to be sent to one other application endpoint. Of course, using a single socket, one could send the same packet to multiple destinations by repeatedly calling **sendto()**. However, this approach has two disadvantages: Excessive network bandwidth is required because the same packet is sent over the network multiple times, and each host must maintain an up-to-date list of all other application endpoints who are interested in its data.

UDP broadcasting provides a partial solution to these issues by allowing a single transmission to be delivered to all applications on a network who are receiving on a particular port. This approach is particularly useful for small networks. When a VR application participant starts, it can simply start listening for data on the application's well-known port and start broadcasting its own data on that same port. It does not need to notify anyone of its presence, and, consequently, the participant can also terminate at any time.

For large networks or Internet-based systems, IP Multicasting is more appropriate (Section 5.1.4).

To send a broadcast message, the sender generates a pseudo-IP address representing the set of hosts that should receive the message. This address is formed by "masking" all of the host addresses that should receive the broadcast with all 1's. By carefully constructing the address, the broadcaster can control the ***scope*** or range of the broadcast. For example, to send a broadcast to all hosts on the 10.25.12 LAN (that is, all hosts with IP addresses 10.25.12.*), the application would direct the broadcast to the address 10.25.12.255.

Using a similar approach, one can send a broadcast to all subnets within a company's network. For example, if a corporate network is network 10, then sending to 10.255.255.255 causes a site-wide broadcast.

For most purposes, however, broadcast is only recommended on the local LAN.

Instead of computing the appropriate broadcast address, one can simply use the address 255.255.255.255. Packets sent to this address will only be delivered to hosts on the local LAN.

Broadcasting is identical to unicasting UDP/IP datagrams, with two exceptions.

First, the destination address must be set to the appropriate pseudo-IP address for the broadcast:

Second, before data is broadcast using the UDP the socket, the application must first register its interest in broadcasting on that socket by calling the **setsockopt()** function with the **SO_BROADCAST** flag.

Once set, the **SO_BROADCAST** option remains in force until it is changed. The actual **sendto()** call is identical to the unicast case.

Broadcast packets are delivered on any UDP socket that is waiting for packets on the corresponding port. Note that a single UDP socket may therefore receive both broadcast and unicast packets, and there is no way for the application to determine whether a particular received packet was broadcast or unicast.

5.1.4 UDP Multicast

UDP broadcasting can only be used in a LAN environment, and even there, it is relatively expensive because each host on the LAN must receive and process the packet, even if no application on that host is actually interested in receiving the packet.

Multicasting is the solution to both of these concerns. It is appropriate for Internet use, as well as LAN use.

It also does not impose burdens on hosts that are not interested in receiving the multicast data.

IP addresses in the range 224.0.0.0 through 239.255.255.255 are designated as multicast addresses. However, the 224.*.*.* addresses are reserved for use by the management protocols on a LAN, and packets sent to the 239.*.*.* addresses are typically only sent to hosts within a single organization. An Internet-based VR application should therefore use one or more random addresses in the 225.*.*.* to 238.*.*.* range if it is employing UDP multicast..

The sender transmits data to a multicast IP address, and a subscriber receives the packet if it has explicitly been set up to listen on that address.

When sending multicast data, an application can specify the IP Time-To-Live (TTL) field to control how far multicast packets should travel. The TTL field can take on values between 0 and 255, and different ranges of the TTL field are designated to represent different network scoping. Though an application can send all of its packets with a high TTL, it is generally desirable to only use the scoping range that is actually necessary. Note that because applications choose multicast addresses randomly, a particular application might receive multicast packets transmitted by some completely unknown application located in a completely different part of the Internet. It is therefore often a good idea to tag your packets in such a way that your application can tell whether a packet belongs to your system.

Overall, multicast is rapidly emerging as the recommended way to build large-scale VR Environments over the Internet. It provides desirable network efficiency while also allowing the Application to allow different groups of receivers, or send certain types of data to different clients by using multiple multicast addresses. They can be used for clients announcing themselves to the network, or requesting data about the terrain they just entered.

On the other hand, multicasting does have some limitations, generally related to the fact that it is a new protocol.

Many older routers are not capable of handling multicast traffic and have to be bypassed by using the Multicast Backbone (or Mbone) to "tunnel" the multicast packets. This is accomplished by packing the multicast UDP packet into a normal IP packet, and forwarding it to the next router, which unpacks it and sends it on it's way again.

Sending multicast UDP packets is almost identical to sending unicast packets. You send to a multicast address instead of a unicast one. One difference is that you should set your TTL value using **setsockopt()** with the **IP_MULTICAST_TTL** flag. You do not need the **SO_BROADCAST** flag set at all for multicast.

To receive multicast data, the application needs to subscribe the socket to one (or more) multicast addresses. The subscription is registered by calling **setsockopt()** with the **IP_ADD_MEMBERSHIP** flag as seen in Listing 5.

Multiple subscriptions may be registered on a single socket by calling **setsockopt()** multiple times. However, most multicast implementations limit each socket to a maximum of 20 subscriptions. If more than 20 subscriptions are needed, the application can create and **bind()** multiple sockets to the same UDP port. To do this, the application must first override the operating system’s security checking that would otherwise prevent multiple sockets from binding to the same port simultaneously by using the **SO_REUSEADDR** flag to **setsockopt()**.

```
struct ip_mreq joinAddr;
       // Adding a multicast IP to the listen set.
       // Specify which multicast address to join
joinAddr.imr_multiaddr.s_addr = inet_addr("245.8.2.58");
       // Specify which local IP address will actually do the multicast join
joinAddr.imr_interface.s_addr = INADDR_ANY;
setsockopt(sock, IPPROTO_IP, IP_ADD_MEMBERSHIP, &joinAddr, sizeof(joinAddr));

/* . . . */
       // Dropping a multicast IP address from the listen set.
       // Specify which multicast address to drop
joinAddr.imr_multiaddr.s_addr = inet_addr("245.8.2.58");
       // Specify which local IP address will actually do the multicast drop
joinAddr.imr_interface.s_addr = INADDR_ANY;
setsockopt(sock, IPPROTO_IP, IP_DROP_MEMBERSHIP, &joinAddr, sizeof(joinAddr));
```

Listing 5 – Adding and deleting IP Multicast memberships

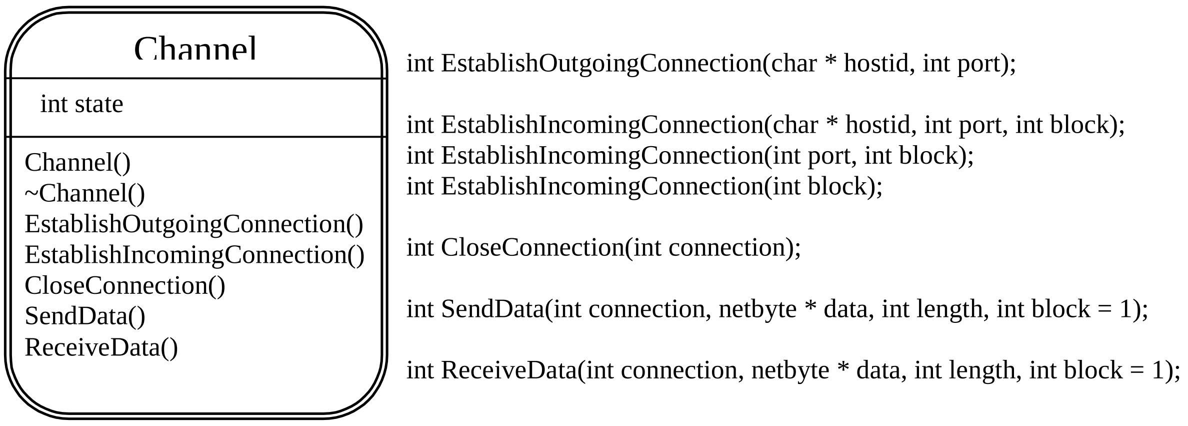
It is generally a good idea to make this call before binding any multicast socket, even if fewer than 20 group subscriptions are required. This call allows other applications on the local machine to use the same port for receiving their multicast data whether on the same or different multicast group addresses.

To cancel a multicast subscription, the application makes a corresponding **setsockopt()** call with the **IP_DROP_MEMBERSHIP** parameter as can be seen in Listing 5.

When a socket is closed, all of its open subscriptions are automatically dropped.

5.2 Channels And NetworkComponents

Channels are an Object Oriented wrapper around the standard BSD sockets. With the generic Channel interface, we can access TCP, UDP, broadcast, or multicast with the same ease.



```
int EstablishOutgoingConnection(char * hostid, int port);

int EstablishIncomingConnection(char * hostid, int port, int block);
int EstablishIncomingConnection(int port, int block);
int EstablishIncomingConnection(int block);

int CloseConnection(int connection);

int SendData(int connection, netbyte * data, int length, int block = 1);

int ReceiveData(int connection, netbyte * data, int length, int block = 1);
```

Figure 8 – The Structure of a Channel

As you can see from figure 8, all channels assume that you can address any system using a machine name and port. This works well enough for TCP and UDP, although the UDP Channel implementation has to provide a pseudo point-to-point structure.

The Broadcast channel ignores the Machine name, and can only be used for sending (the standard UDP channel is used to receive broadcast messages) and the Multicast channel requires the Machine name to be a Multicast address (225.0.0.0 → 238.255.255.255)

As discussed before, sending multicast UDP is identical to sending unicast UDP. This makes it possible to use the standard UDP channel to send to multicast addresses, but

would still require a different approach for receiving multicast data. For this reason, I decided not to combine both but to provide a separate multicast channel (the MCastChannel) which deals specifically with multicast data. This also allowed me to set the multicast Time to Live setting (the **IP_MULTICAST_TTL** flag). Using the **IP_ADD_MEMBERSHIP** flag discussed in section 5.1.4, you can receive from as many as 20 (the limit on some systems) multicast addresses. By binding the port multiple times with different sockets, this can be extended indefinitely. This allows us to refine the VR design further, by allowing components to listen to only the messages that interest them. By assigning different message types to different multicast addresses, we can effectively filter the data seen by a component.
NPSNet used this technique to extend the size of their simulations, they assigned hexagonal areas of the simulation world to different multicast addresses, in this way they controlled how much a unit could see. It also allowed units to move around in a large environment without being flooded by all the messages from units that could not even affect them. [Macedonia et al 1995]

5.3 Network Server Component

My first attempt at implementing the NetworkServer was not as successful as I had hoped, the system had severe drawbacks and performance problems. I had attempted to make the NetworkServer just an extension of the Channels, basically a container class for a number of channels. It worked fine for the testing I had performed on it, but when I decided to try it in a full application, I noticed that it did not interface with the rest of the system as well as I had hoped. The system had been creating and maintaining it’s own channels, and the only way for an application to send data to a specific channel was to keep a list of all the channels that had been opened. This was easy enough if the application had actually initiated the connection via an **OpenConnection()** call, but if the connection had been initiated by the server (from a listen) the application then had to poll the server to get it. I did not consider this an easy to use system, and decided to redesign the server component.

The second design also was not exactly what I wanted. It required the application to create the channels and pass them to the server. The server would then manage the created channel, but since the application had created it, it could refer to it without recourse to the server. This design, while looking good on paper, turned out to be more unwieldy than the first design, I found it easier not to use the Server in the system and handle the channels directly.

My third design turned out to be a far more robust design than the first two, while not being as generic. Since the UDP channel could be used to receive from more than one source, I decided that it did not need to be included in the server structure, and built the server specifically to handle TCP network connections. This freed me from having to use the Channel components and allowed me to redesign from the ground up without having to rely on compatibility issues.

In my redesign, I looked carefully at what I wanted the server to be able to do, and realised that extending the standard Channels would not accomplish my design goals. The NetworkServer had to be able to handle its network connections by itself, without needing the application to create and maintain them. It also needed to be able to interface more closely with the VR system. I moved the Server up from a simple component to a VRComponent, which now gave it some added abilities. These abilities, being the ability to discern between different VR data types meshed in well with the Server's ability to make decisions on data compression.

The server uses low level TCP communication, so it can multiplex the channels better. It accepts connections to a port and then binds a socket to that port as shown in the TCP section (Section 5.1.1) and continues listening on the port. In the **ThreadRoutine()**, the server checks for data on a socket and feeds the data into the corresponding VR Port. It also checks each port for outgoing data and relays that to the corresponding TCP socket.

The server also supports separate calls that are not linked with the (almost) automatic handling of the VR Ports.
I decided for the application to decide how many connections it wanted to support and then pass that in when it created the network server. This design still requires the application to take care of linking the VR Ports to the right objects. With the inclusion of the Environment Interface, however, this could be minimised.

5.3.1 Data Compression

The data compression routines use two different levels of compression, besides none. The first is a RLE type compression algorithm of my own devising, which works very well with repeating data (raw graphics streams, and some data types.) and a PKZIP type compression provided by the zlib library.
Due to the zlib library only being available on the Linux machines, I have that section #ifdef'd out if it is compiled in IRIX. This can be changed when we install zlib on the IRIX machines.
The server tells if a stream is compressed by reading the first three bytes. If they start with a 'R' and a 'Z' then the stream is compressed. The third byte determines the compression scheme used. The compression routine will not perform compression if the compressed stream would be bigger than the uncompressed one.
The compression is switched on and off from the application. I had tested earlier versions with status monitoring, but the system was very unstable, and would switch between compression and no compression randomly, causing strange effects.

5.3.2 Network Status Monitoring

This was removed from my final server due to problems with the algorithms. I had attempted opening a separate UDP channel to a client and sending timestamped packets once every few seconds. If the response from the client indicated that there was significant latency on the network, the system would begin compressing the data.

The type of data I was testing with limited the usefulness of this technique. The data did not compress well, and so caused a higher CPU load on the machine with little or no increase in network performance due to compression.
I decided that it would be better left to the application to decide what data would compress well and therefore save CPU time on the server.

6. Results

I have split the results section up into channels and server. I will summarise what was achieved with each in terms of efficiency and the ability to better scale the system up.

6.1 Channels

I implemented a Broadcast channel and a Multicast channel to provide the system with the ability to send data streams to more than one client without the need to instantiate a peer-to-peer link for every client.

6.1.1 Efficiency

Efficency was improved dramatically with both channels, And some interesting data came to light in testing. I was testing the maximum number of packets the system could process in a specified time with each of the channels. Table 1 illustrates my results.
The experiment used 64 byte packets, and attempted to force the network card to transmit them as fast as possible. I did not check to see if the packets had actually been transmitted.

Table 1 – Packet Creation Rates

Data Type	Transmit Rate	Standard Error	Data Rate
UDPChannel	6860.78	32.29	3.5Mbps
BCastChannel	6843.56	37.14	3.5Mbps
MCastChannel	36577.80	75.11	18Mbps

The surprising result was the Multicast channel, which was creating packets six times faster than the UDP or the Broadcast channels. This may be due to the fact that the Multicast protocol requires less setup than the other two, but I'm not completely convinced that these results are valid. More testing with different hardware and testing receive rates for the machines would also be in order. If the results are validated, it would be another point in favour of using Multicast over either of the other two channels. From the Data Rate measurements, you can see that even at full speed, the broadcast and unicast channels cannot even send enough packets to flood the network. The Multicast channel, however was creating packets at almost double the network bandwidth rate. I had unplugged the network from the machine, so I expected packet rates above what the network could handle.

6.1.2 Distribution potential

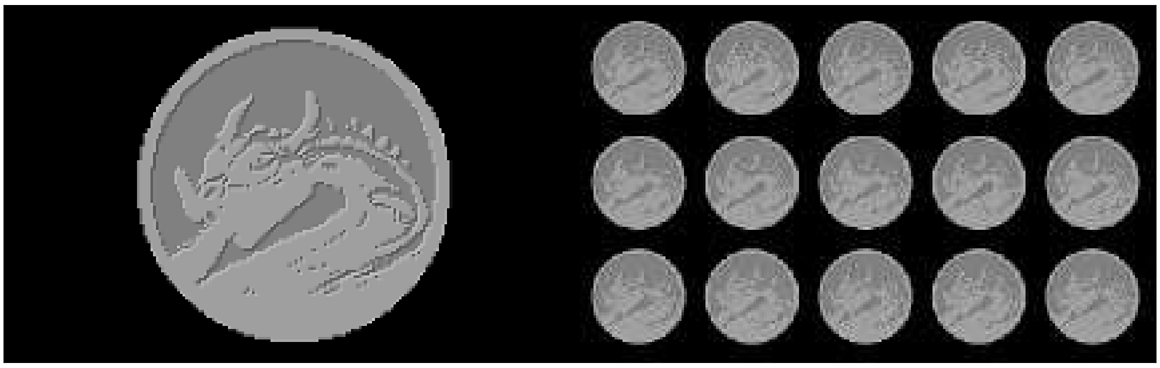
Using the multicast and broadcast UDP protocols, we can extend the effectiveness of the CoRgi system by a large amount compared to the old method of a connection per client. The effective bandwidth needs grow far slower in the multicast case than in the unicast one. While the limit has been extended, however, it hasn't been entirely eliminated. Good Environment design, and a good distributed design will also help in this area.

6.2 Server

The Server was created to cater for a reliable stream oriented system. It would be well placed for sending graphics streams to "thin" dients, but does not have as much potential as the UDP Multicast channel architecture.

6.2.1 Efficiency

The server instantiates a peer-to-peer link for every client attaching, and therefore is not very efficient in network terms. However, for large reliable streams, it has the ability to compress them before sending, thus reducing the network bandwidth requirements. For graphics based streams, this compression can be very efficient, due to the properties of computer generated graphics.* In a simple test, I passed a 256 colour graphic into the system, consisting of a single graphic object (figure 9) and a



Figures 9 and 10 – Single and Multiple objects

second graphic, with multiple similar objects. I used the RLE compression algorithm and the zlib compression on the pictures, and came up with the results in Table 2.

Table 2 – Results of compression test on two graphics.

Graphic	Uncompressed	RLE	ZLib
Single Object	64770 bytes	4398 bytes	1880 bytes
Multiple Objects	64770 bytes	24163 bytes	2186 bytes

In both cases the RLE and zlib algorithms achieved significant savings on the graphic data. Using a packet size of 1.5K, the first picture would only require two packets in its compressed form, and the second would require two in zlib, and the RLE version would require 17. Uncompressed, both pictures would need 44 packets to transfer.†

* Most computer generate graphics tend to have large areas of the same colour, and do not smoothly change over the entire picture. Photographs, on the other hand tend to have smoothly varying areas of shade, which is why JPEG compression works better on photographs.
† The GIF version of the figure 9 was 2626 bytes, coming in between the RLE and zlib algorithms.

6.2.2 Distribution Potential

The NetworkServer gives us the ability to plug in a simple TCP based client – server system. It is limited by the number of sockets allowed on a port, and also the bandwidth required.

7. Conclusion

I attempted to provide a networking subsystem for CoRgi, without compromising on configurability, adaptability and efficiency. I studied a number of other Distributed VR systems and noted their advantages and disadvantages, and how they related to the CoRgi design. I provided a design that I thought would not compromise the CoRgi design goals, but would provide for any eventuality that might occur in subsequent CoRgi design decisions. I then implemented a set of networking systems which can be used the the CoRgi system, with an aim to provide a number of differing subsystems which could be plugged together to create a networking system of arbitrary complexity. The components I implemented have shown the ability to increase the efficiency of the network usage, and have provided us with a set of tools that can be used to model most of the current networked Virtual Environments currently in existence. This will allow us to test different network designs against each other in a controlled environment, with the minimum of variable changes.
I believe I achieved my objectives to design and implement a set of efficient networking components for use in the CoRgi VR system.

7.1 Future

Future work with the system could entail implementing the Envinronment Interfaces, without which the NetworkServer component does not perform as well as it should be able to. There needs to be a standardised packet format for CoRgi interactions, possibly based on DIS, but far more extensible. Once the packet format and a distributed architecture is decided upon (possibly using [Mclean 1997] as a mould) we

can begin to test the networking potential to it’s fullest, and test out new configurations using the CoRgi design of having pluggable components.
CoRgi’s extensible design lets us design completely differing architectures and test them in a simulation, which could add significantly to the field of Networking in Virtual Reality.

References

1. [Burka 1995] Burka L.P., *The MUD Timeline,* http://www.apocalypse.org/pub/u/lpb/muddex/mudline.html, 1995.
2. [Walsh et al 1995] Walsh, A.A., Jancke, A., Bandeira, F.D., O'Brien, F., Soares H.I.G, and Nikolaou, S., *Where Reality Blends with Fiction,* MSc/Diploma in Electronic Information management coursework (http://www.shef.ac.uk/uni/academic/I-M/is/studwork/groupe/home.html), 1994/1995.
3. [Brutzman 1997] Brutzman D.P., *Graphics Internetworking: Bottlenecks and Breakthroughs,* Chapter 4, *Digital Illusions,* Addison-Wesley, Reading Massachusetts, 1997, pp. 61-97.
4. [Leigh et al 1997] Leigh, J., Johnson, A., DeFanti, T., *Issues in the Design of a Flexible Distributed Architecture for Supporting Persistence and Interoperability in Collaborative Virtual Environments.* In the proceedings of Supercomputing '97 San Jose, California, (http://www.evl.uic.edu/cavern/cavernpapers/sc97/index.html) Nov 15-21, 1997.
5. [Roehl 1995] Roehl B., *Distributed Virtual Reality -- An Overview,* (http://sunee.uwaterloo.ca/~broehl/distrib.html) First draft: June, 1995
6. [Macedonia et al 1995] Macedonia, M.R., Zyda, M.J., Pratt, D.R., Brutzman, D.P. and Barham, P.T., *Exploiting Reality with Multicast Groups: A Network Architecture for Large-Scale Virtual Environments,* Proceedings of the IEEE Virtual Reality International Symposium '95, North Carolina, March 1995.
7. [Macedonia 1995] Macedonia, M.R., *A Network Software Architecture for Large Scale Virtual Environments,* Doctoral dissertation, Naval Postgraduate School, Monterrey, California, June 1995.
8. [NCSA 1998] National Center for Supercomputing Applications, *Caterpillar Inc. Distributed Virtual Reality (DVP) Project,* http://www.ncsa.uiuc.edu/VEG/DVR/, 1998

9. [Mclean 1997] Mclean C., *Multi-User, Interactive Virtual Reality via the Internet*, Honours Thesis, Computer Science Department, Rhodes University, November 1997

10. [Leigh et al 1997] Leigh, J., Johnson, A., DeFanti, T., *Issues in the Design of a Flexible Distributed Architecture for Supporting Persistence and Interoperability in Collaborative Virtual Environments*, In the proceedings of Supercomputing ' 97 San Jose, California, Nov 15-21, 1997

11. [Johnson et al 1998] Andrew E. Johnson, Jason Leigh, Thomas A. DeFanti, Maxine D. Brown, and Daniel J. Sandin, *CAVERN: The CAVE Research Network*, Paper at the 1st International Symposium on Multimedia Virtual Laboratory. Tokyo, Japan, March 25, 1998

12. [Curtis et al 1994] Curtis, P and Nichols, D.A., *MUDs Grow UP: Social Virtual Reality in the Real World*, Proceedings of the IEEE Computer Conference, IEEE Computer Society Press, Los Alamitos California, 1994, pp. 193-200. Available at ftp://ftp.lambda.moo.mud.org/pub/MOO/papers/MUDsGrowUp.ps.

13. [Cowcroft 1998] Cowcroft, J., Handley, M., Wakeman, I. Internetworking Multimedia. Book to be published by UCL press. 1998. Available at http://www.cs.ucl.ac.uk/staff/J.Crowcroft/mmbook/book/book.html.