Boston College
Computer Science Department

Senior Thesis 2003
Frank Mazzacano
A Reliable Multicast Protocol for a Distributed System
Prof. Robert Signorile

0 Abstract

Since the advent of Internet gaming users have been mostly concerned with a quick connection that will allow them to receive and send game state information. However, users also demand that the information they exchange is done so in a reliable way, to maintain fairness between all clients. This ended up making developers weigh pros and cons in picking the packet transportation protocol. They settled on a unicast protocol, which is more reliable, but not as quick as possible. I have chosen to use the less utilized protocol of multicast to send and receive information between clients in a distributed gaming system.

The purpose of this project is to integrate a three-dimensional model of a real-world environment into a distributed system based on multicast communication between nodes in the system. The development of three-dimensional environments (3DEs) revolutionized the computer graphics industry, in particular the gaming industry, by bringing computer graphics one-step closer to replicating reality. The developed application uses a 3DE of a dormitory (specifically Ignacio Hall on the Boston College campus) to serve as a backdrop for a distributed system in which users compete against each other for resources and objectives (such as health powerups and to obtain a winning score). The development of a distributed mutual exclusion algorithm operating via multicast communication was necessary to ensure the proper performance of the developed distributed system. The created algorithm takes advantage of the characteristics of a multicast network and various synchronization mechanisms, such as a timestamp and the election from the multicast group of a central arbitrator. These

synchronization safeguards not only enforce mutual exclusion, but also aid in the enforcement of reliable communication between nodes in the distributed system. The implementation of a dynamic rotating server selected from among the members of the multicast group, in combination with a timestamp applied to all messages sent between nodes in the system, forces the ordering of messages sent within the system. This adds a level of predictability and stability to developed system and its communication mechanism. The final implementation and integration of the 3DE, the distributed mutual exclusion algorithm and the reliable system based on the multicast communication protocol result in a distributed, three-dimensional world, which mimics the everyday interaction between independent people vying for shared resources.

1 Introduction

The purpose of this project is to create a reliable multicast protocol that will be used to transfer game state information over a distributed system. The distributed system that is used will be that of a simulated 3-D environment for a computer game modeled after Ignacio Hall on the Boston College campus. Typically information of this sort is transported using a structurally reliable protocol, such as unicast. However, in this project I will be focusing on using the predefined multicast protocol with additions to make it more reliable. The enhancements added stem from a dynamic rotating server that is utilized to grant permissions and maintain order. Together, with the mutual exclusion work done by Jonathan Pearlin and the graphic design completed by Evan McCarthy, we constructed the end result of a totally distributed gaming system over a multicast network.

1.1 Unicast Protocol

Most of the gaming protocols that exist today are designed with a unicast system. A unicast system is one in which there is, "one sender and one receiver."[1] Game developers choose this protocol in order to ensure reliability without losing too much time from speed. A client in the game will directly connect to the server and send him the information that he wishes to spread. The server then is in charge of distributing this information to the rest of the clients in the game. Because this information is transported point to point there are various handshaking methods that can be used to ensure the information has been correctly received. The server is also able to distinguish which player is trying to grab a resource first, according to when the connection is first established. This adds yet another layer of fairness to the game in which the server is allowed to force the ordering of events in a fair, first come first serve manner. One fallback of a TCP is the handshaking and error detection process adds additional travel time to each packet, slowing down the game state transfer from client to server and then to all other subsequent clients. I chose to use a different protocol called mulitcast.

1.2 Multicast Protocol

The multicast protocol is one that is radically different from the more traditional unicast one. The main difference with a multicast design is, "applications can send one copy of each packet and address it to the group of computers that want to receive it."[2] The major advantage to this is the client must only send out one packet, and then

[1] Kurose and Ross, "Computer Networking: A Top-Down Approach Featuring the Internet".
[2] Cisco Corporation, "Multicast Routing"

each other client that is part of the multicast group will receive it. This means that there is no intermediate steps and bypasses necessary to slow the transmission of the packets down. One necessary element to this protocol is special routing hardware that is necessary for the packet to get split into more packets while getting sent out to the rest of the clients. You must realize that some routers do not have the capability to do this, hampering the ability for this protocol to work over a widespread area. The most essential pro of using a multicast protocol is the speed. Because only, "the sending of one packet from one sender to multiple receivers with a single operation"[3] is needed you cut down much of the handshaking and rerouting time done by a traditional server.

1.3 My System

The system that I designed used the predefined multicast protocol with additions to make it more reliable and fair. The major development that is added is the idea of a rotating, dynamic server. This server is only used for a portion of the packets that are being sent out that will necessitate permission to do certain tasks. Because the server only receives this small portion of packets the rest of the game state transfer is much faster. The server also has mechanisms built in to make sure that something a client says happened did actually happen. The major breakthrough of my work is based on this dynamic server, and the ability of it to rotate from client to client being sure not to overload the system of one with the burden of extra work.

1.4 Putting it Together

As you know, my portion of the project is only one section of a three-part project done by myself, Evan McCarthy and Jonathan Pearlin. Evan McCarthy focused on

[3] Kurose and Ross, "Computer Networking: A Top-Down Approach Featuring the Internet".

the graphic portion of the project, in which it was my job to send the distributed material of it from client to client. Jonathan Pearlin focused on the deadlock prevention and mutual exclusion of the code to ensure that it was not a faulty system.

2 Previous Work

Before trying to create my protocol and system I did a great amount of research on what already exists and what people have proposed to do in the future. I found many interesting ideas and topics about multicast gaming, rotating servers and the idea of using a timestamp to ensure order in the system.

In trying to implement a system in which a multicast protocol is used for the transportation of packets there are a three things that the developer must worry about in trying to keep the system fair and reliable. A great summation of these pitfalls can be found in work done by Garcia-Molina and Spauster. The first thing that one must be weary about is single-source ordering. This has to do with one source sending two messages, say m1 and m2 to the group. The group must receive these two messages in the correct order, that being m1 first, then m2 second.[4] What must be understood here is that if one client is trying to send out two packets in near the same amount of time, they will be received in the correct order. In my system the rotating server is used to prevent the disordering of any important packets that may be hazardous to the entire group, even if the server has changed in between the two packets.

[4] Garcia-Molina and Spauster, "Ordered and Reliable Multicast Communication."

The second worry that one must have when developing a reliable multicast system is multiple-source ordering. This is the idea that if two sources send out two packets in the order of m1 first and m2 second, all of the clients attached to the groups must receive them in the order of m1 first and m2 second.[5] Once again I use the rotating server to fix this problem and ensure that all essential packets are sent and received in the proper order; this will be described later.

The final circumstance that might alter the game and the reliability of it is multiple group ordering. This is much similar to multiple source ordering, but there is one caveat added. If the packets m1 and m2 are sent in that respective order they must be received by the processes attached to the group also in that order. This even includes the situation where they are sent to different, but overlapping multicast groups. This is a problem that I do not have to specifically worry about, due to the fact that the clients may only be attached to our group.[6]

Related to the above information, I then had to find a way to make sure that we could order events to make sure that they were indeed sent/received in the correct sequence. To attack this problem I read up on work done by Lamport. She gave me great insight on what to do to fix this. The idea came in the form of a timestamp. One must realize that, "Synchronization between the clocks kept by each system is imperative to avoid unexpected results."[7] To follow this I added the timestamp to each packet. The server mainly utilizes the timestamp. The server will be the only one that has the ability to

[5] Garcia-Molina and Spauster, "Ordered and Reliable Multicast Communication."
[6] Garcia-Molina and Spauster, "Ordered and Reliable Multicast Communication."
[7] Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System."

update or change the timestamp in order to keep things reliable and not granting any of the individual clients too much power and authority. The timestamp used is that of a logical clock. This means that it does not actually change by a timing mechanism built into the code. Instead it is based on events. This does indeed keep up with Lamport when she stated, "Clock correctness is not based on physical time, but rather the order in which events occur."[8]

After I established a good foundation of research based on the idea of ordering events and how to make it happen on a multicast network, I tried to find some information on the plans for a rotating server. I was not able to find much work on it that has been done, but I did find one interesting idea on the ordering of the server rotations. In a work done by Charikar, Halperin and Motwani there was the proposal of actually having more rigid guidelines of basing a k-server mechanism to rotate on. Their k-server mechanism is not exactly the same as the one I propose. They have it so all the clients may either be a server at the same time, or not be one at all. Whereas I propose to only have one of the clients be a server, and the rest of strictly slaves to it. When it is then the servers time to rotate it is done based on timestamp considerations. In the k-server proposition they add a whole new level of dictating who rotates next. They have the idea of making the server rotate to the next closest one to save physical time in the rotation.[9] I think this is a good idea, one in which would be a good add-on to my system. The only problem is it would have been highly unlikely to develop this in the amount of time needed and being able to

[8] Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System."
[9] Charikar, Halperin and Motwani, "The Dynamic Servers Problem."

add it to my system. It seems to be a whole new thesis on it's own. Other than that work I was unable to find any other published information on rotating servers.

3 Implementation

3.1 Windows Multicast Connection

Without previous knowledge on how the Microsoft system of networking worked, it was necessary to first find out how to establish and maintain a multicast connection using their predefined methods for the C++ language. All of the work that is needed to make the connection is found in the constructer MulticastSocket(). In this constructer you need to do a few things to bind the user to the group. The first thing that is done is initializing Winsock. This is simply done with preexisting Microsoft code. After that it is necessary to first open up a datagram socket and change the address of the socket. The code to do this is as follows:

```
UDPSocket = socket(AF_INET,SOCK_DGRAM,0);
addrLocal.sin_family = AF_INET;
addrLocal.sin_addr.S_un.S_addr = htonl(INADDR_ANY);
addrLocal.sin_port = htons(MGROUP_PORT);
```

After this is done you have to do a sequence of steps to ensure that you may continue to use this socket to send data. The first step to this is making sure that it is indeed reusable. You do not want the socket to close after you use it once. The single line of code to make it reusable is:

```
setsockopt(UDPSocket,SOL_SOCKET,SO_REUSEADDR, (char *)&reusable,
sizeof(reusable)
```

With the socket being reusable you next want to bind yourself to the group. The reason to bind the socket to the group is to receive the messages. The only way for

your computer to receive them is to establish its socket as one of the ones that will receive from the group. To bind your socket this code is necessary:

```
bind(UDPSocket, (struct sockaddr*)&addrLocal, sizeof(struct sockaddr)
```

Now that the receive end is taken care of I have to make sure that you can send packets out. This has two parameters to it. First, you have to be able to send packets from the computer. Secondly, the packet needs to be multicast so that the router will be able to send it to all the other receivers that are bound to the group at the time. After actually joining the group you have to adjust the time to live value. The time to live value is what decides how many different hops the packet will make before it is ceased being sent. A hop would be each time it made it to a different layer or router in the network. I chose a time to live value of 8 that we put into the constant TTL. The code to actually join the group and then to set your time to live value can be seen below:

```
setsockopt(UDPSocket,IPPROTO_IP,IP_ADD_MEMBERSHIP,(char*)&mreq,sizeof(mreq)
setsockopt(UDPSocket, IPPROTO_IP,IP_MULTICAST_TTL,(char*)&TTL,sizeof(TTL)
```

The last part of the code that is necessary to be sure that the connection is properly made and you will be able to send is adjusting the address destination. This means changing a few variables in a structure much like what we did for the receiving above. The code for this is:

```
addrDest.sin_family = AF_INET;
addrDest.sin_addr.S_un.S_addr = inet_addr(MGROUP_ADDR);
addrDest.sin_port = htons(MGROUP_PORT);
```

The final thing that I decided would make my protocol and system easier to maintain and manage was to turn loop back off. Loop back is usually on for a multicast socket. What it means is that when you send a packet out, you will also receive it in your

receiver buffer. I chose not to use this since it would add a lot of code that would need to check if it was yours, and if so disregards the packet. The code to turn loop back off is as follows:

```
setsockopt(UDPSocket,IPPROTO_IP,IP_MULTICAST_LOOP,(char
*)&loopBack,sizeof(int)
```

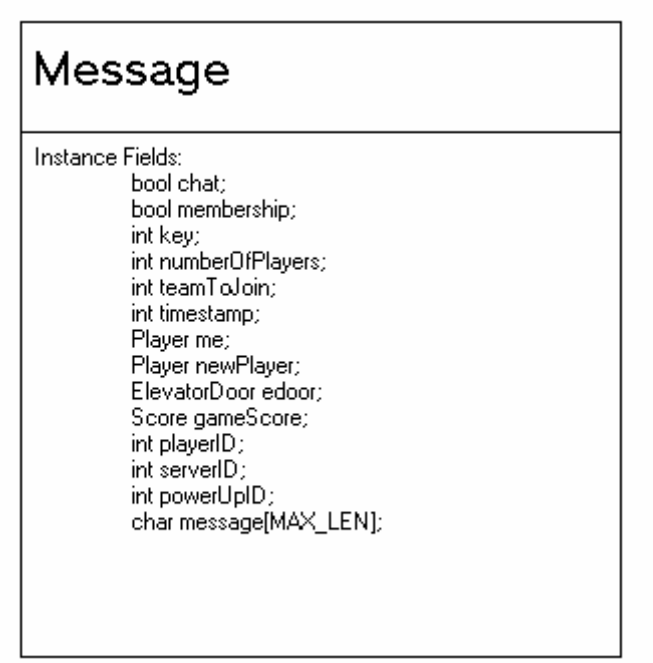
Now that the connection is established as one that you may receive and send multicast, or datagram packets on, I needed to worry about a higher level of control. When the entire project is put together we will need to have not only the display drawing, but also the communication on the network, so we need threading. To make my part of the project threaded I put in two different threads. The first is a receive thread and the second is a send thread. There is also the need to make sure that the threads will not be changing the same data at the same time, so locks are needed to keep control of the threads. This is a major part of Jonathan Pearlin's project, as he developed the locking scheme, so data and information on it can be found in his thesis. It is also important to note that the receive thread is only awaken when indeed it has received a message, and the send thread is woken by main when it determines that a new game state packet must be sent out.

One of the things that I learned is how to deal with the multicast connection in windows and C++. This is a big reason that we chose C++ to do the project. We knew that if we used java the connections were much easier to establish and use, for I have had experience manipulating them before. However, the graphics would have suffered immensely from the virtual machine. We also wanted to do this project in

windows, which has its own form of Winsocket. If we were to use Unix it again would have been much easier for me to do, because in other classes I have not only done multicast networking on Unix, but also have dealt with threading extensively.

3.2 Message Structure

The whole multicast system is based on the messages that are sent from client to client throughout the game. To try to enhance the use of the Message object I created I made it so the message is always the same one sent. So, whether the message is for chatting or for just a normal game state you are always sending the same object. However, flags in it represent the difference between each message. There is basically a hierarchy of messages in the system. The highest layer has three different types. There is a message used for chatting, for membership and for game state. To switch in between each of these messages there are methods defined in Message.cpp that are used to toggle the flag depending on what you wish to send. I also have a method that is very important in keeping the data relevant in each packet. There is a method called clearMessage() in Message.cpp. This method is used to clear out all of the fields of data in each message. This way if you sent a chat message previously and the data you wanted other clients to see is still in it, it will not be falsely propagated down the line. I simply zero it out so that is not possible. There are also the obvious helper methods included in Message.cpp. This means there is a getter or setter for each of the instance fields the class has. All of the instance fields that are contained in each message can be seen in the class diagram below:



Now I would like to break down which of each of the three major types of messages are used for. The first one is a membership message. This is the type that is sent at the beginning of the game when the client first wishes to join the game. He sends out a membership message with the key equal to the constant JOIN (the initial handshaking procedure will be explained in depth in section 3.5). A membership message with key equal to JOIN is also sent whenever anyone wants to join a game that already exists. There are three other usable key values with a membership message. The first is OK_TO_JOIN. This is sent out when the server decides it is legal for a new client to join the gaming environment. The other clients also send this type of packet out to the new player once the server has notified them to do so. The second type of key that may be used is the GAME_FULL key. This key is rather self-explanatory. This is what is sent out when the server receives a request from a client to join the game as a player and the server cannot allow them to because the

game is already full. When the client receives this the program simply ends. The final type of key that you are able to use is the LEAVE key. This is the key that a player sends out when he wishes to leave a game. Each player then deletes that player from their player list and makes the necessary updates to their other variables, such as number of players.

The second major type of message that may be sent out is a chat message. The idea behind the chat message is very simple. While the player is moving around he may hit either the 'T ' key to chat to his team, or the 'R' key to type to everyone. The player then types what he wants sent and hits 'Enter'. When the 'Enter' key is hit the message is ripped from the input and put into the message that will be toggled to a chat message. He then sends this packet out. When the other players receive a chat message they take out the message that was typed and send it to display. From display the message is printed out on the screen.

The third major type of message that can be sent is a game state message. The game state message has many different types of sub messages. The first of which is a SYNC. The sync message is sent out when a server is going to be synchronizing with the next server on the list. This packet is sent out, and the new server responds with a message of subtype RESPONSE. During this interaction the timestamp is making sure it is synchronized between the two. The next subtype is PING. This message is sent out when a client becomes the server for the first time. The server sends this to make sure that all computers are active and are capable of becoming the next server.

If a computer has frozen and they are allowed to become the server the game would freeze as any message they need to approve or deny is received. The next type of key is just a plain GAMESTATE. This packet is simply sent anytime a player moves or does anything involving input from the mouse or keyboard in most circumstances. Another type of key that can be seen is a HEALTH packet. This means that a player has been hit and the server is sending out a packet with the adjusted health value. The second to last type of game state is that of key SHOOTER. This type is sent out when the server has determined that someone has been hit by another. The server sends out a message denoting who the shooter was and who got shot. The final type of message can be that with a key equal to DEAD. This means that someone has been shot and killed. When a player receives this the new packet will have the updated score, and the player that was killed knows he has to restart. The value of all of these constants can be seen in the chart below:

Constant	Value	What it means
CHAT	0	Message is meant for chatting to other players
JOIN	0	Client wishes to join game.
LEAVE	1	Player is leaving game.
OK_TO_JOIN	2	Client is being granted permission to join.
GAME_FULL	3	Client is rejected because of full game.
GAMESTATE	1	Sent with general movements
REQUEST	0	Sent from old server to synchronize
HEALTH	4	Someone has been hit and health is being adjusted
DEAD	5	Sent when someone has been killed
SHOOTER	6	Sent when a player has hit another player with laser
RESPONSE	7	Response to current server to synchronize
SYNC	8	Sent to current server to synchronize
PING	9	Sent when client becomes new server

3.3 Timestamp

In this section I would like to explain the idea of the timestamp that is incorporated into the multicast scheme in our program. Because I am trying to target a multicast protocol that is much more reliable than a traditional one where there is not control at all, I settled on the necessity for a timestamp. This timestamp is a logical clock. As you should remember this is not a timed clock, but one that changes based on a sequence of events. With the use of the timestamp the server will be able to tell whether or not certain events that have a mandatory placement on order have happened in the right order. For example, if someone claims that they have shot someone but the packet in which this information is sent states that the timestamp was one that is older than the servers current timestamp the server knows to discard that information, for it is not accurate. This is also extremely useful in people trying to pick up some kind of resource, such as a health pack. If a player tries to pick one up after the timestamp they have has expired, the server knows not to grant them permission to pick it up. The timestamp is also used when a new player joins the game. They know the player is new because the timestamp will be of the value one and because they are sending out a membership message with the key being JOIN.

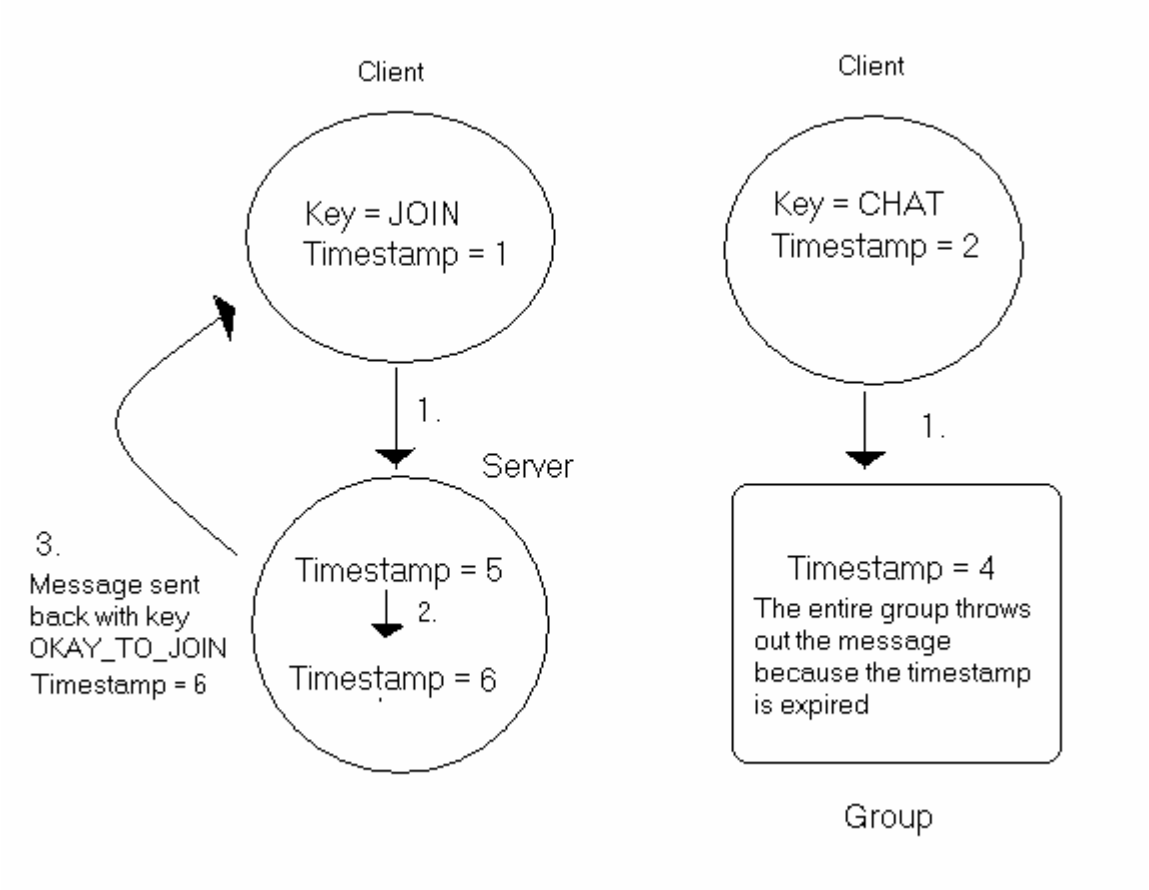
Another important part of the timestamp is that only the server has the ability to manipulate it. It is solely the job of the server to change and correct the logical clock whenever a certain type of message is received. Some of the messages that are server dependant are ones that grab resources, ones that have players either entering or leaving the game, and ones that have a player shooting another one. Anytime any of these events happen the server will update the timestamp before he sends out the

packet that will notify all the other players of this event occurring with his guarantee that it was indeed a legal move. When the players then get this message with a higher timestamp than the previous they will change theirs to reflect the new one.

In the current game the timestamp is set to not continue to climb forever. The limit of the height of the timestamp is based on the number of players in the game. The game is made to allow each player (only when they are the server) to update the timestamp ten times. So if there are 3 players the maximum timestamp will be 30. When all of the players see that the timestamp has hit the maximum they know to set it back to zero, where the player in position zero will be the new server. Details on how exactly the server rotates will be discussed in section 3.4. It is important now to realize that each of the players will be using the timestamp to directly decide when they should start to be the server and when their duties as the server are over. There will be no problems with the delay that might happen when servers are switched for two reasons. First, the switch should be instantaneous, and secondly if a packet is sent and no server gets it the player will just resend it. This might cause some lag as common in most networked games.

An example of how the timestamp works can be seen in the figures below. In the one on the left in the first step a client sends the server a packet with timestamp=1 and the key=JOIN. This means that he is trying to join the game. The server recognizes this and sends him an OKAY_TO_JOIN packet back with the timestamp updated by one, so it is now six. Everyone else gets this packet also, which is not denoted in the

figure. In the figure to the right a player sends the whole group a message with key = CHAT. However, the timestamp=2 in it. The rest of the group is up to timestamp = 4, so they disregard the packet and the chat message is never seen.



3.4 Rotating Server

The most important aspect of the multicast protocol I designed is the rotating server. Because early on I decided that the protocol needed to be as reliable as possible I needed to come up with an idea to keep order. After some serious thinking I decided that a rotating server would fit my scheme perfectly. One of the reasons I chose to use mulitcast instead of unicast was I wanted it to be totally distributed amongst all the players in the game. I did not want one independent server that took control over

all of the parts of the game. I did not want the players only to be able to connect directly to the server and communicate with him. So by having a server implemented on a multicast network, I am able to not only enforce order and reliability, but I am also able to maintain the abstract idea of having no real server in the system.

The server that I am using is rotating. By this I mean that the server is always rotating from player to player. This means that not one computer can say that they are in charge of the game as a whole. This is also extremely valuable in terms of not hurting one player too much. If the server was stuck to a player and static the whole time that person would be taxed an extreme amount dealing with requests. That computer would always have more things to compute and more jobs than the other computers, not giving him a fair chance to play. So by having the server move from one person to another I evenly distribute the problem of having computation slowed a bit over everyone on the network. This is much more fair then what is typically done in a unicast system. This way everyone has a chance to be the server, and everyone also has the number of players – 1 time to be totally on their own.

Another great aspect of the rotating server was the ability for it to grant permission and enforce the rules. When I originally had the system designed with no server, questions came up depending on what happens if some computers are attempting to do illegal maneuvers. Who is in charge of recognizing he is cheating? Who realizes how to stop him and fix what he has done? These were all problems that I was encountering. So by having the server there I do have a higher authority with the

power to decide what is indeed right and wrong. This adds a whole new level of security to the system, as it does not allow players to do whatever they wish to do. The server is also able to make sure that the player's requests are ordered correctly. The first person to get their request to the server is the first person to get that request taken care of. This agrees to the principles of multiple source ordering as given by Garcia-Molina and Spauster[10] It is very important to have this aspect fulfilled to keep the game as fair and ordered as possible.

The first thing I wish to talk about is how a client determines if they are the server or not. The way that the players do this is by looking at the timestamp. Before and after any major change in the game, such as a person leaving or entering, the players run a sequence of code that is meant to figure out at what timestamp they are to become the server, and at which timestamp they are to cease being the server. The code looks as follows:

```
while(!foundMe){
    Player* temp = list->getPlayer(i);
    if(temp->getId() != me->getId())
            (*myPlace)++;
    else foundMe = true;
    i++;}
if((*myPlace) == (list->getNumPlayers() - 1))
    *I_AM_LAST = true;
else *I_AM_LAST = false;
*start = MAX_SERVER_TRANS * (*myPlace);
*stop = (MAX_SERVER_TRANS * ((*myPlace) + 1)) - 1;
```

As you can see from the above code, first the player has to find his current spot in the player list. He knew it before, but he has to recalculate it because someone might

[10] Garcia-Molina and Spauster, "Ordered and Reliable Multicast Communication."

have entered or left changing what place he is respective to the other players. Once they have found their new spot they calculate their stop and start timestamp. They also check to find out if they are the last person in the list, which affects the resetting of the timestamp. So by running this code a player is able to always know when they are and when they are not the server. This prevents an unfortunate situation where two different players think they are the server at the same time and try to grant permission to the same things.

Another important part of the start and stop timestamp recognition is the SYNC messages that I mentioned above in the message section. When the server begins to realize that he is about to relinquish his duties as the server he goes into a chunk of code that is utilized to have a handshake with the next server. He sends them a SYNC message that starts the handshake. Through it the server basically confirms to the next person what he already knew, which is that he is the next server. However, another important part is a timestamp leap I implemented. I had it so that the server first calculates whether or not the next server is adjacent to him in the list. If not, the server will figure out how many void spaces there are between him and the next. He will then increment his timestamp accordingly. So if there are two inactive servers between him and the next he would increment the timestamp by 20, skipping the two inactive servers. This means the timestamp can leap tens of numbers between one exchange. This makes the synchronization of the timestamps between all the players much more simple to all the players that are involved. The code to do this can be seen below:

```
*timestamp = (*timestamp) + ping->getTSIncrement();
*I_AM_SERVER = false;
if(*timestamp == ((list->getNumPlayers() * MAX_SERVER_TRANS) - 1))
    *timestamp = 0;
else if(*timestamp > ((list->getNumPlayers() * MAX_SERVER_TRANS) - 1))
*timestamp = ping->getTSIncrement();
```

The second thing that I want to talk about as pertaining to the rotating server is a neat little gimmick I added to it. After deliberation I decided that it is possible that the person that is supposed to be the next server has gone idle, or his computer has frozen. In this circumstance it is possible that he has not left the game, but is still technically available to be the server. If this were to happen when he is tagged the server the game would cease to work and chaos would be afoot. To solve this problem I came up with an inactivity timer. The job of this timer is to prevent an inactive server from taking control. The way that I worked this was when a player first becomes a server he goes into a special loop. He sends out a special packet with key = PING to each player. When each player gets this they respond to the server saying they are active. The server then places an active flag in the list of players. The code to send pings is as follows:

```
if((*timestamp) == (*start) && firstTimeToPing){
    ping->resetPingData();
    sendMessage.clearMessage();
    sendMessage.setServerID(me->getId());
    sendMessage.setMembershipStatus(false,0);
    sendMessage.setChatStatus(false,0);
    sendMessage.setKey(PING);
    sendMessage.setPlayer(*me);
    sendMessage.setTimestamp(*timestamp);
    mSocket.sendData(&sendMessage);
    ping->setPlayerActivity(me->getId(),ACTIVE);
    firstTimeToPing = false;}
```

When the server sees that it is about to lose its power it checks its inactivity list. If the next player to become server never responded to him with an active response he will not let him become the server. He will prevent this by incrementing the timestamp enough to cover the potential frozen servers work time, thus skipping him. The server will update the timestamp enough to find the next active server. And if the next one is before him in the list, it will indeed reset the timestamp to the proper number.

Finally, let me get into the actual duties of the server. The first actual duty of the server is to allow new players to join the game. Because we are able to utilize the server as an authority on the game as a whole, we have something that can tell a new client if they are able to join. So when a new player sends out a packet that is requesting to join the game the server first receives this. Every other player in the game is coded to disregard a join message. When the server decides it is okay for that player to enter he sends the appropriate OKAY_TO_JOIN message out to that person. Each other player then receives it and knows to send out their own OKAY_TO_JOIN to that player. A second thing that the server is in charge of is when a player shoots another one. This scope also includes when a player dies, or the game is over, because the server has the last say on what affects the actual scoreboard. When the server receives a game state update he first checks to see if that person has shot his laser. If so the server compares this to each person to see if any of the lasers have hit anyone. If so it is the server's job to send out the appropriate SHOOTER messages stating that someone was hit, and or killed. Another important thing that the server is

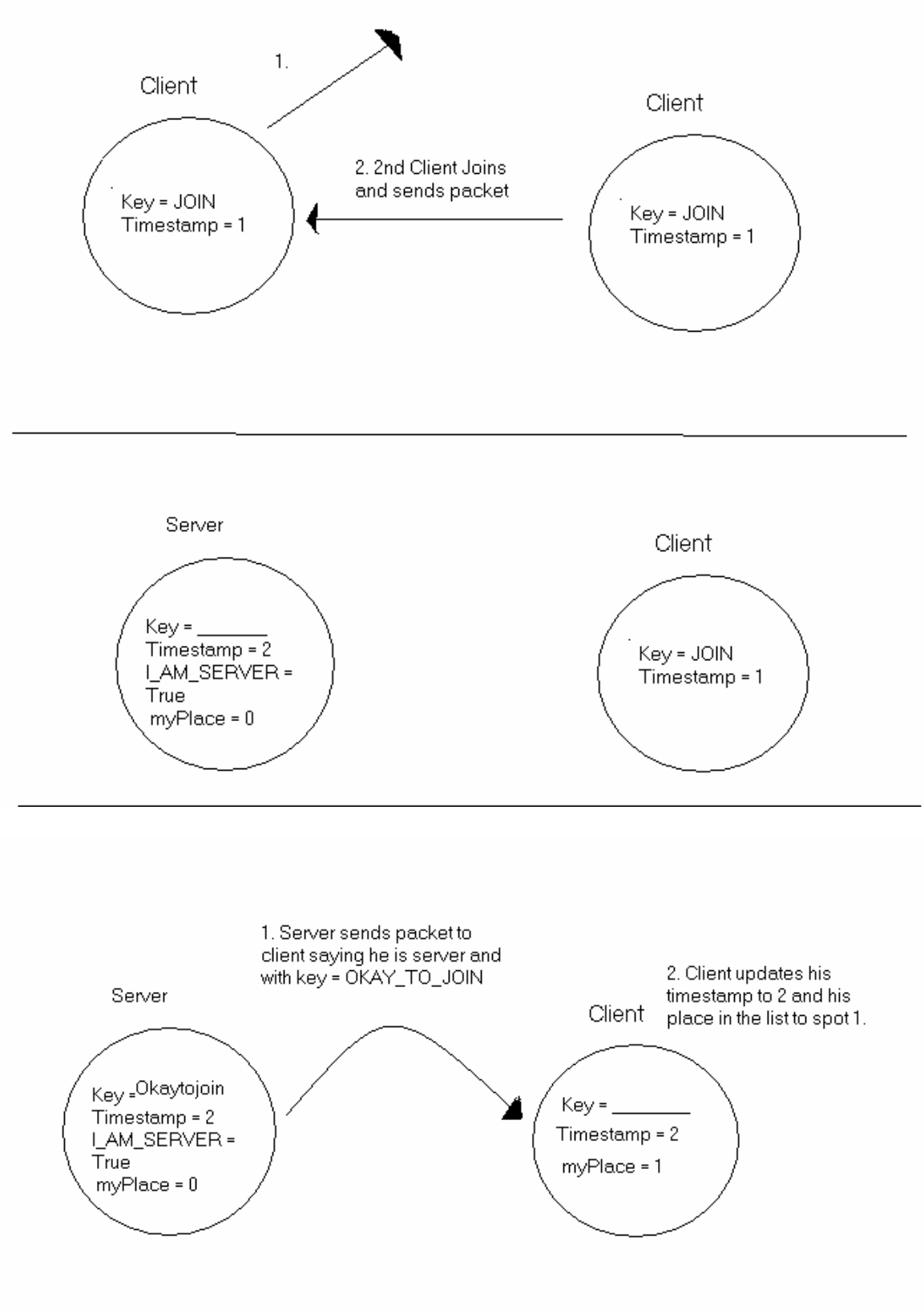
in charge of is to make sure that only one person can pick up a health item at a time. If two people try to do it in a very close matter the server is in charge of making sure the person that sent out the message first is the one that is granted permission to take it. The server's final job is to make sure that it transfer's correctly if the server quits the game. The next person in the list automatically becomes the server and takes over the duties for him.

3.5 The Game's Beginning

Now that I have given the essentials to the theory of how the multicast protocol will work I would like to get into some more specific details of the actual game's creation. Before there is anyone trying to play there is nothing that exists on the multicast group. When the first person joins he has his key set to JOIN and his timestamp to one. When this packet is sent there is no one attached to receive it, so it is simply lost. Now the real magic starts to happen when the second person tries to join. That person also has his key set to JOIN and his timestamp to one. He then sends this message out, but it is not lost. The first person that joined receives it. When he sees that it has the same information as him he knows that there is no game yet and that he is in charge of creating it. So he takes the information and then makes himself the server. He then changes the timestamp by one, to two and sends out an OKAY_TO_JOIN packet to the other player. The other player receives this and realizes a few things. First, he sees that there is now a player that has assumed the role of the server. He also gets the number of players in the game from that packet, which is two. He also sees that he must now set his id to one, for the server has already set his to zero. He changes all of these values that he has to and is now able

to compute his own start and stop time to become the next server. So when this is all done we have the knowledge of the server, the correct timestamp, each players start and stop value for becoming and losing the rights to the server and a base to finally start playing the game on. At this point the world that Evan McCarthy designed will begin to draw and the game will be in full force.

One item of interest is that the server code is unique for the first person in the game. Only the first person in the game will ever run the code that is there, and that is it. However, the other client's code is much more portable. Every other player that tries to join the game will be able to run it. I will explain this in section 3.7, which talks about what happens when additional players try to join the game. A diagram that displays the initial handshake between the first two players can be seen below:



3.6 Gaming with Two Players

Once the game has been created with the two initial players the game can begin to work in full swing. This means that all the parts that Jonathan Pearlin, Evan McCarthy and I worked on will work together. The two players will be able to move around as they wish in the 3-D environment like they will be in any other game. They will originally spawn in the base of the team color that they chose on the initial input and game start screen they see. Every time the player moves either the mouse or presses one of the arrow keys they will send out a new update packet. When the other player receives this he simply updates that player's information in his list and redraws it with the necessary changes. When a player shoots, an update is also sent every time the laser moves another spot. This is to make sure excellent collision detection can be done by the server to tell if someone was hit.

In a two-player game the server does not have as many jobs as he normally would. First off he only has to decide if either his laser hit the other person, or if the other person hit him. Another duty is to check to see who picked up the health packet first. The server also has to watch out for the other player quitting. He just has to make sure that he deletes them from his list. The server only has to ping one other player to see if he is active, and he most likely will be. The server's only other real job is to mediate when one of the two dies. If someone dies he simply has to send out the correct packet and that will reset the player's position. If the game then ends the server starts the timer to respawn every one and the new game will begin. Included in this is the server's job to make sure he updates the scoreboard after every death.

The last thing that will occur often during a two-person game is the chatting aspect. As I have stated earlier there are two keys that a person can hit to send a chat message. When the player hits one of the keys he will then be able to type his message. When done the player will hit 'Enter' to send it. In a two-player game if the players are on different teams and the player chooses for only his team to see it, no one will print out his message. However, if he chooses all the players to view it or if the other player happens to be on his team he will see the sent message. All of these are aspects that make up a fully 3-D networked game.

3.7 Adding Additional Players

Now that I have established what events will occur when two players are in the game at the same time I will tell you what happens when any other additional player tries to join. Once there are two players in the game the server will be the only one that respects a message with the key set to JOIN and timestamp of one. When he sees this he does a few things. First, he will check to see if the game is full already. If it is full he simply sends the client a GAME_FULL message, causing him to terminate his program. If the game is not full the server next moves onto checking the team. If there is room on the team the client wished to join the server will let him join it. If there is not room on the team the server will automatically find a remaining team that has space for him and will force him into that spot. With all of this done the server will then place that users information into the packet and send it out with the key OKAY_TO_JOIN. The server then goes through the loop that was looked at above that will calculate his new start and stop time for being the server.

When the other players in the game receive this packet that has the key OKAY_TO_JOIN and coming from the server they know that a new player is being allowed to join the game. The first thing they will do is add him to their list. They then know that they have to send out a new packet with their information so the new player can begin to add all of the existing players to his list of gamers. After the player has sent out his information to be processed he makes sure to ignore all the other OKAY_TO_JOIN messages that will be sent out by all the other active players, for they will have no bearing on his game. Finally, that player will have to enter the same loop as the server has to. He first has to determine his place as compared to the rest of the players on the list. And off of that the player has to recalculate his start and stop time for being the server.

Now lets look at what the client that is being granted permission to join does. Once he receives the message with the key OKAY_TO_JOIN from the server he knows that he is now able to become a legitimate player in the game. He first has to rip all of his information from the packet that the server sent him. After that he has to begin collecting data from the rest of the players. To do this he enters the following loop that will only recognize the correct packets:

```
while(i < players-2){
    mSocket.receiveData(&receiveMessage);
    if(receiveMessage.getKey() == OK_TO_JOIN){
            list->addPlayer(&receiveMessage.getPlayer());
            i++;
            tsLock->enterCS();
            if(*timestamp < receiveMessage.getTimestamp())
                    *timestamp = receiveMessage.getTimestamp();
```

```
tsLock->exitCS();}}
```

When the player is done collecting the data from each of the other players in the game he is now allowed to start playing the game like the rest of the players. One thing that he has to do first, however, much like everyone else, is to computer his place respective to the other players and to find out his new start and stop time for becoming the server when it is indeed his time. This sequence of events can occur up until the set maximum number of players in the game have been admitted and started up.

4. Conclusion

After finishing my part of the project I was extremely satisfied as to the outcome of it. After the research I was able to decide exactly what I needed to get off the ground and start a gaming network based on the multicast protocol. There were always questions as to what exactly I would have to do to make it all work and be much more reliable than the things that had been done in the past. It was not overly difficult for me to add onto previous work given the nature of the multicast protocol. You have to remember that not all the hardware on routers on the Internet and networks are multicast compatible. Because of this it has not been a great idea to put a lot of time into the development of games that are run over a multicast network, for there would be questions as to whether or not the game would be portable to all potential users.

The most important part of my work was to get the rotating server off the ground. It was very essential that I could have some higher source of authority that could grant the

various degrees of permission to the different users. Without that the game would not have been run properly. The key parts of the rotating server were the inactivity timer to prevent a defunct computer from trying to take over leadership, the timestamp mechanism to make sure that there was some sort of logical clock that all the players could look to and synchronize their timestamps to, and making sure that the timestamp would go back to the beginning value when the number of transactions for each player being the server had been done. If we did not reset the timestamp it is possible that we can get overflow in a variable when the timestamp gets too high. Once I was able to get the rotating server off the ground I was much more confident in being able to put the whole system together.

None of this could be accomplished however without the work of Jonathan Pearlin and Evan McCarthy. My networking protocol would not have been of too much help if there was nothing to try and test it on to make sure that it was relatively fault proof. Evan's virtual world was exactly what I needed in order to test and run my system to be sure that it was in actuality efficient and fast. I was extremely happy with how quickly it did work and how little it bogs down the network on the computer that was running it. Being able to transport the different degrees of Evan's world was a great thing to have work. It was a great feel to see that my network could effectively transform his single player world into one in which multiple players could walk around and interact without a huge delay. It was also very nice to see how well the lasers worked. When one player would shoot their laser you could easily see the lasers propagate through time on the different players screens. The elevator doors were also much like the laser. Evan implemented elevators

in the game. A player is able to walk up to the elevators and hit the 'Space' bar to open the door. Every player in the game can subsequently see the door open at the same time and the player enter it. The player can then press 'Space' again to shut the door and press the number of the floor they want to go to and they will be brought there. This is yet another great way to see that the game is indeed distributed between all the players.

Another extremely important part of the program was designed by the valiant efforts of Jonathan Pearlin. Jonathan had the much more tedious task of designing the game as whole and making sure that the entire thing worked without getting hung up. Some of the many problems that could occur were from race conditions, deadlock and starvation. Jonathan successfully managed to block all of these possible catastrophes. This enabled the game to work extremely efficiently without having to worry about one of the threads fighting another thread for power or control. Without the work of Jonathan, merging the parts that Evan and I did would have been near impossible.

In the end I am extremely satisfied with the position that we got to in the completion of this thesis. In the beginning there was much doubt that we would be able to put a piece of work to this extent together in under a year. However, we were able to finish it by the due date and we are all extremely satisfied with where it is at. Much of this might be because we are roommates and were able to discuss it so often without a lot of hassle, or possibly because of the fine guidance by our sponsor Professor Robert Signorile. Whatever it may be I leave this exercise with valuable academic lessons that I will remember forever.

Special thanks to Jonathan Pearlin, Evan McCarthy and Robert Signorile.

Sources Cited

Cisco Corporation, "Multicast Routing," http://www.cisco.com/warp/public/614/17.html. March 1999.

Charikar, Moses, Dan Halperin and Rajeev Motwani, "The Dynamic Servers Problem," 1998.

Garcia-Molina and AnneMarie Spauster, "Ordered and Reliable Multicast Communication," August 1991.

Kurose, James and Keith Ross, Computer Networking: A Top Down Approach Featuring the Internet. Addison Wesley. 2001.

Lamport, Leslie, "Time, Clocks, and the Ordering of Events in a Distributed System," July 1978.