Project Number: MLC-MM04

IMPROVING SCALABILITY IN AN ONLINE GAME

A Major Qualifying Project Report

submitted to the Faculty of

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Bachelor of Science

by

_____

Andrew Brommelhoff

_____

Jesse King

_____

Brian Platt

_____

Allen Seitz

Date: April 27, 2005

Approved:

_____

Professor Mark Claypool, Advisor

Abstract: The goal of this project was to reduce the operating cost of a massively multiplayer game by making its servers more scalable. To accomplish this task we implemented two methods for improving the scalability. These two improvements are a peer-to-peer sub-layer and a means for allowing a client to become a temporary server. To test the effectiveness of our two improvements we implemented an original massively multiplayer game. The results from our experiments show improvements in latency and processing load.

1. Introduction

Massively Multiplayer Games (MMG's) are one of the newest types of video games. These video games are different from traditional video games in that they require a scalable network of clients whereas traditional video games feature a limited number of clients or players. MMG's have allowed for the implementation of virtual worlds in which hundreds of thousands of players can all share one game together. This is a desirable feature for players because it creates a dynamic social environment for the players. Many of those who play MMG's extensively prefer to have this social interaction in their game because it enhances their enjoyment of the game in ways that are exclusive to MMG's. For example, a standard feature in MMG's is the ability to chat online with other players. This feature makes MMG's more socially immersive by allowing players to personally contact one another. Another common feature in MMG's is the ability for players to play competitively or cooperatively with whomever they want and with as many people as they want. Sometimes, even in a massive world of other players, a player will choose to play solo. However even in this case the solo player is still aware of the other players and the changes to the game that they cause. This awareness creates a sense of belonging to the game world, which also makes the game even more immersive even though the solo player is neither chatting nor playing with other players at all. Another feature that is common in many MMG's is the ability to create custom avatars. This grants players a higher degree of freedom and allows them to create a character the way they want them to look. Whether the player's avatar is a representation of how they see themselves deep down or is a character purely out of their

imagination, character customization adds an element of uniqueness to each player and variety to the game world.

One of the technical challenges that must be solved in order in implement such a system is the problem of determining how much data should be sent to each client and how often that it should be sent. Solutions to this problem have been shown to vary depending on the type of game. If too little information about the current state of the game is sent then a player might be missing information that they need to play the game. If data is not sent often enough to each client then each client will lose his or her valued awareness and the game will be less fun. However, over-sending data to each client has consequences as well. If a client receives too much information, either all at once or over a period of time, then that client might be unable to process it all and, thus, the game's performance could suffer. Some clients also might not be able to handle the network strain that a MMG could cause. Striking a good balance between sending more information less often and less information more often would be ideal and it would let the largest number of players play with optimal performance. However, this balance is often impossible to achieve because of the static requirements of the game's design. The global game state can almost always be divided into parts that can be classified as either critical to each client or non-critical to each client and then the easiest thing to do is to send only the critical data to each client. While this strategy is the safest in that it protects the integrity of the critical data, it does not take into account any optimizations for cases when normally critical data might be irrelevant. Optimizing a game system for those

occasions is an important topic because and this MQP will implement at least one such optimization.

An alternative architecture to the client-server model would be a peer-to-peer (P2P) network. P2P networks have considerable benefits in that they are scalable and the decentralized structure of a P2P network makes it more reliable than a centralized client-server model: one peer going down will have little impact on a P2P network, but a server going down could crash the entire game world. While these gains in scalability and reliability are fairly significant, P2P networks lack in the areas of security and are not well suited for handling persistent data. The differences between peer-to-peer networks and client-server architectures will be covered more in-depth in Section 2 on MMG Architectures.

For this project we focused on making a massively multiplayer game more scalable while still using a client-server model. We had two reasons for doing this. The first reason is because client-server is the most commonly used system in modern online games. The second reason we chose this model is because we found that this system has advantages over other similar systems and that its weaknesses, specifically the server bottleneck, could be worked around. Our goal with this project was to find ways to relieve some of the load on a central server. If certain data that is unnecessary for the server to process (such as chat messages, avatars, etc.) were handled solely between peers, the load on the server would be reduced, thus improving scalability.

Since we required that a game be easily modifiable at the network level we implemented our own small game from scratch. This game supports a number of clients that are only limited by system resources, the ability to send text messages to other players who are currently signed on, and a few other aspects of a fully functional MMG, such as combat with other players.

One of the optimizations that we made was the establishment of a peer-to-peer sub-layer network for the transmission of text messages. This is in contrast to a system where each text message must go through the central server. The reason for doing this is because only the central server knows the IP addresses of each client and therefore, it must be ultimately responsible for routing each conversation. However, performing these routing duties must be scheduled with other game tasks and it is possible for this load to be a significant burden on the server. With the addition of a P2P network, the game clients wishing to send text messages only use the central server to locate other clients and then they can communicate directly with each other. We believe that this will have the biggest positive impact when two players chat somewhat continuously during a long play session, which is the norm for a massively multiplayer game. Some modern MMG's such as *Phantasy Star Online* use "voice chat." This feature allows players to verbally communicate each other using a headset microphone. Although this feature fell outside of the scope of our project, it would be quite possible for future work to modify our P2P implementation to support the transmission of voice chat messages.

Another optimization that we made is to allow the server to forward requests for certain pieces of game data to other clients. This optimization was applied to static, persistent data, such as the distribution of a patch to each client from the central game server during a game. Much like the text messaging problem, patch distribution is another burden that a game server must typically bear while trying to operate a video game. However, in this case the danger of the server becoming a bottleneck is much greater. The size of a patch for a MMG can be very large (several megabytes or more) and when this is multiplied by the current number of players, the result could be a catastrophic spike in latency for all users if other precautions have not been taken. Our changes have reduced this bottleneck by allowing clients who have finished the download to act as hosts for other clients. The only circumstance under which this may happen is when the server determines that it is suffering from a critical overload and responds by requesting that an able client be the host for another client who would otherwise be unable to download the patch. If for any reason the download fails, which could be due to the connection being lost or because the game-appointed server refused to serve, then the player who was unable to download initially, continues to wait for another download to begin from the central server (possibly this time coming from the central server itself).

Having the clients share non-static, non-persistent data in the same way could also be used as optimization; however, this presents a game security issue that can be difficult to solve. Even though it is possible for the server to remember which clients it has asked to serve data to other clients it might be difficult to utilize this information in real time. When downloading something large and static, such as a patch, if a player were to

somehow cheat and manipulate the patch before it is sent to another player then it would be possible to detect the change and for the server to correct the change and take action on the cheater. However, this is most likely too difficult to do in real time. Instead of pursuing this option, we have identified which parts of the game state are non-critical and we have suggested ways that clients could still be used to share this information. These details will be covered over the course of this report.

In the next section, we will analyze and evaluate the primary architectures used in massively multiplayer online games as well as any research that has been done in this field. The third and fourth sections discuss program and experiment implementation and the data obtained from the experiments. Sections five and six will cover analysis and conclusions that can be drawn from our data. The final section discusses possible future work and enhancements to our project.

2. Background

2.1 MMG Architectures

When designing a Massively Multiplayer Online game, there are a number of possible architectures to choose from, each with their own strengths and weaknesses. Here, five network models will be examined and discussed in terms of four attributes: scalability (how easily the architecture can handle additional clients with respect to the amount of resources needed to accommodate them), reliability (how well the game world can recover if a server goes down), security (how well the architecture can detect and

deal with cheating), and persistence (how well the architecture preserves the state of the game world and/or client data when a client leaves or comes back).

On one end of the taxonomy spectrum (refer to Table 1 at the end of this section) is the simple Peer-to-Peer (P2P) architecture. Here, all computers act as both clients and servers at all times. Scalability is best here, since there is no single point to which all data is routed, which could otherwise create a bottleneck when many players are involved. Similarly, reliability is excellent, since a player dropping will not cause the other players' games to end due to the decentralized structure. (Refer to Figure 1 at the end of this section for a detailed taxonomy of centralized vs. decentralized online gaming networks). Security is purely on the honor system; any player at any time could give themselves an unfair advantage over the other players and since there is no mediator to keep players in check, this is a significant problem in some game situations. Lastly, there is no persistence with a basic P2P design.

On the opposite end of the spectrum from P2P is the basic client-server architecture. Here there is one server to which all clients connect. This server acts as an "all-powerful, all-knowing entity" through which all gameplay data goes. This results in a drastic increase in security and offers the possibility of persistent data. Conversely, however, this architecture alone is poorly scalable since the server can become the bottleneck and a single server is not very reliable either.

A step up from the basic P2P is the addition of super-nodes. Each computer still acts as a client, but rather than having everyone also be a server, only certain computers are servers at one time, though every computer is a possible candidate to become one. The super-node computers are sporadically changing. Scalability is still excellent for the

same reasons. This P2P architecture does offer some persistence as long as the super-nodes stay up; however, this also introduces slight problems with reliability – if each super-node goes down, the game will be unable to continue. This architecture only has some advantages over basic P2P, primarily in the areas security. While P2P with super-nodes does offer some form of cheat detection, those who are super-nodes could potentially abuse their power or look the other way with respect to cheating.

A third type of P2P is Peer-to-Peer with "trusted nodes." Again, all computers are clients, but only "trusted" computers can be servers. This offers advantages in security and persistence, since the "trusted" computers are actual people hired by the company running the game. (Let us assume that these people are indeed trustworthy.) This, in turn, makes the architecture less scalable than the previous P2P architectures, since the addition of more clients would require more trusted nodes. Lastly, reliability is a bit lower, since the servers are limited, as they are always the same people, and, despite being "trusted," servers can always unintentionally go down.

Towards the other end of this spectrum is the Distributed Server Farm. In this case, we have multiple computers, all servers, working together as a distributed system. All other computers connect to this system as clients. This architecture has a great deal more capability in terms of security and also persistence, since all data can be saved to the servers. However, with a lot of data going through the server, this creates a bottleneck and the need for additional servers may be necessary, especially if the MMG becomes popular. Reliability is also not high, since again, servers can always unintentionally go down.

Name	Scalability	Reliability	Security	Persistence
Peer-to-peer (P2P)	Excellent	Excellent	"Honor System"*	Abysmal
P2P w/ Super-nodes	Excellent	Good	"Honor System"	Poor
P2P w/ Trusted Nodes	Good	Good	Fair	Fair
Distributed Server Farm	Fair	Fair	Excellent	Excellent
Client-Server	Fair	Fair	Excellent	Excellent

* "Honor System" indicates that security *is* possible within this architecture; though, the players can cheat (or commit some other insecure activity) just as easily as play fair

Table 1: A table classifying each architecture in terms of scalability, reliability, security, and persistence. As the architectures become more centralized, they become less scalable and less reliable. However, the more centralized architectures are more secure and allow for persistent data.

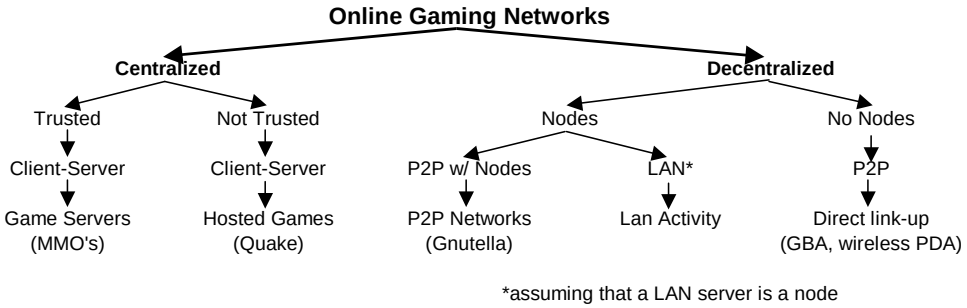


Figure 1: A taxonomy of online gaming networks by category. The less centralized an architecture is, the less trusted the network can be. Like in Table 1, Nodes in a P2P network can be either trust or non-trusted depending on the network, but they are certainly not as reliable as a dedicated server. A LAN Server is more dedicated than a Node in a P2P network, but typically does not have the resources to handle large numbers of clients.

2.2 Previous Work on this Topic

MMG's have been an area of research interest lately among academics and professional game developers alike. Often times, however, the research done will ignore one or two of the primary aforementioned attributes (scalability, reliability, security, or persistence) in order to maximize the others. Two particular articles, one by Yahn W. Bernier [Ber 01] and another by Takuji Iimura, Hiroaki Hazeyama, and Youki Kadobayashi [IHK 04] propose some practical scalability solutions, but are largely applicable to peer-to-peer architectures. Nonetheless, they introduce some techniques that would be possible to modify and incorporate in a primarily client-server architecture.

Bernier begins his paper by giving a background of how the client-server architecture of online video game functions. He says that the clients are essentially "dumb": all they do is take input given to them by the user, send this information to a server that performs some operation on it per the game code, and then renders the reply from the server. The bottleneck, Bernier continues, is in the actual network connection. The lag is the client sending the data to the server and waiting for a response. To the user, their character appears to stutter, commands inputted are not instant. He suggests that to solve this lag, the programmers could have the clients predict how the server will respond and render it. Thus, to the user everything appears instant and when the actual response from the server arrives, the game updates to agree with it. Player movements, attributes, weapon stats, and displaying of opponents are the critical things that the clients will predict. The benefit of this is that there is no apparent lag for the user. The biggest drawback, however, is player "warping". If your client predicts an opponent at a location

in space and the server predicted it somewhere else, when the reply from the server arrives at your client, the opponent will appear to warp, or instantaneously move, from the old location to the new one. Another con he describes is called “leading”. If you shoot at a moving opponent directly with an instant hit weapon and are running with a high lag. By the time your data has reached the server, he or she has moved out of range and the server registers a miss. So in order to hit them, you need to lead your shots. However, based on your lag, the leading required to hit something could be very large, like one hundred units in front of the target. Bernier continues with the introduction of a possible solution to leading. Lag compensation is logging when a user command is sent and the server rewinds the time of the opponent to match the lagged user command time so that instead of registering a miss it would be registered as a hit. Bernier adds that this too can cause problems such as “bullets around corners”. If you shoot at someone, a stationary target, in your line of sight and your lag to the server is great enough that they can run around a corner and duck. No matter what their client predicted, they would register a hit and would possibly die. Thus their reaction would be, “that bullet killed me from around the corner, that’s impossible”. Bernier concludes with a warning that this however will change the game play and that may not be what the programmers original intended the game to be like.

Iimura, Hazeyama, and Kadobayashi describe how to use a Distributed Hash Table to adapt an MMG to a peer-to-peer network. Each peer keeps this DHT, which contains the global state of the game and the contact information of its peers. To handle larger sizes the game is broken down in zones, so that the DHT only has to hold the global information of its zone. To handle timing issues a peer is selected to be a zone

owner. This zone owner acts as the server of its zone. For performance the zone owner caches it's DHT so that it can make many updates quicker. Also for network performance, connections are cached so that volatile data, like player position, can be updated frequently. To handle the effects of having an owner of a zone being removed, the system they have allows any one of the peers to have the ability to promote itself to be the new zone owner whenever they do not find an owner of their own zone. This can be done because the game state is save on the DHT, which is shared amongst every player.

3. Procedure

This section of the report describes the process that we used to complete the project. First, in Section 3.1, we list the goals of our project. In this section we explain each component of the project and why we needed to create it. Section 3.2 is dedicated to describing the three programs that we wrote in order to get our results. In Section 3.3 we describe our experimental procedure and how our results were obtained using the three programs.

3.1 Goals

When we started the project, we analyzed the strengths and weaknesses of the most common server architectures as they relate to massively multiplayer online gaming. The three main architectures that we studied were a client-server model, a typical distributed server model, and a pure peer-to-peer model. Two additional variations of the peer-to-peer model (the use of "super-nodes" and "trusted nodes") were studied as well. Each of

these models was then compared to each other using the attributes of scalability, reliability, security, and persistence. Our goal here was to determine if the most commonly used architecture, the distributed server, was really the best-suited model for implementing an online game. We discovered that this model is actually very appropriate for online games, but it could also benefit from scalability improvements. This work is presented in section 2.1 of our report.

Next in our goal was to prototype and test a client-server model with enhanced scalability. To come up with ideas on how to improve scalability, we looked back at our analysis. The peer-to-peer networks were an obvious choice for improved scalability. However, we did not want use them in such a way as to reduce the system to a peer-to-peer network since it would compromise the client-server model's strengths in security and persistence.

Some game traffic is unimportant to accurate gameplay and has a very low need for security. Our solution was to implement a peer-to-peer sub-layer for game traffic that peers could use for unreliable or temporary messages. We also implemented a means for peers to send larger, more permanent updates to each other. It is not a problem that the server does not know what is sent between peers, since these events do not affect the rest of the game. But with the updates that are critical to the game state, each client will need to verify the integrity of the data that it has downloaded with the server. Since this is game data, the player can verify the data easily by simply playing the game. If during the game, the client’s software shows that the data it received from another peer is out of date or severely corrupted, then the game server can take action. In the worst case, the server will just have to send the update itself. But in the best case, which is the most frequently

occurring, the data will be valid and the game will be playable. The explanation of the implementation of our system is in Section 3.2 of our report.

Finally, we ran the programs several times in controlled experiments to determine how effective our modifications were in improving the scalability of the server. Since measuring scalability is difficult to define, we decided that our goal would be to show that our improvements should decrease the processing load and bandwidth load of the server in most games. Then, if the expense-per-client could be lowered, then a single server can handle more clients than it could before. This statement is what we mean by improved scalability. Section 3.3 describes exactly how our experiments were run. The results and the analysis of those experiments are also included in Section 4 of this report.

3.2 The Program Specification

This section of the report describes in detail how our system was designed and implemented to meet our goals. In Section 3.2.1 we begin by explaining the rules of our scaled-down massively multiplayer game. Section 3.2.2 explains the implementation of our basic client-server system. Section 3.2.2 also shows how our improvements work on top of this implementation. In Section 3.2.3 we explain the purpose of the MapServer program and we also explain why it is decoupled from the client-server system. In Section 3.2.4 we define each of the game events that are sent over the network, since the bulk of each program's work is in listening for and responding to these events. Finally, Section 3.2.5 describes the different versions of the server and client programs that we created just for the experiments.

3.2.1 Program Rules

When we began designing our game, we had three important requirements for it. The first was that it had to be a truly an MMG. This meant that hundreds of players could be involved in the same game world at the same time. The second requirement was that it must contain the same basic functions as a complicated MMG. For our project, we wanted the players to perform all of these functions: entering the game, leaving the game, walking around, talking to other clients on the same map, attacking other players or enemies, and changing player state. Finally, our most important requirement was that the game itself not be very complicated to implement and play. This requirement was imposed by our limited time to complete this project.

To meet these requirements we designed the game "The World of Rock, Paper Scissors". This game meets exactly all of our requirements and no more, so that the project could be kept simple. To play, a player must first log in by starting their client program. (The opening screen is shown in Figure 2). Next, the server places the client in a random, unoccupied place in the game world. This is the player's starting position. The player can now take any of these actions: log out by quitting the program, type a chat message by pressing the “Enter” key, take a step up, down, left, or right by using the arrow keys, or change their weapon by pressing any of the “r”, “p”, or “s” keys. There is a 400-millisecond delay between steps. This number was chosen since it seemed to be a reasonable speed for the game to be played as it gives a player time to react to other players in their proximity. However, there is a weapon-switch delay of 100 milliseconds.

The reason for this is so that the players with fast reflexes can switch their weapon in mid-step when engaging in combat.

Each player is also given a unique avatar in the game world. This is what the player controls when they issue a walk command. Also, each avatar has a weapon icon displayed above it. This weapon icon is always either a picture of a rock, a pair of scissors, or a piece of paper.

The meaning of these symbols is used when two players collide with each other. When that happens, both players are returned to their previous position. However, they also engage in a game of “Rock, Paper, Scissors” (hence the name) using the weapon that they currently have displayed as an icon. The loser of this combative game loses one hit point. Players start with 15 hit points so that they can enter combat a few times before running out of hit points. (Figure 3 shows some players who have been engaging in combat). When a player runs out of hit points they receive a loss and they are moved to a new random place in the game world, exactly like if they had restarted the game. The player who took the loser's last hit point is awarded with a win and continues playing on the same map. Players retain their win-loss records until they log off. The combat symbol and a player's win-loss record both meet the requirement that the players have a state that changes during the game.

The game ends when the server closes and, thus, is not persistent. When this happens, the state of each client is lost. This is not what a commercial MMG would do, but it is acceptable for ours because persistence is not one of our goals. Persistence would be less useful in testing our network since we restart the game anew for each trial.

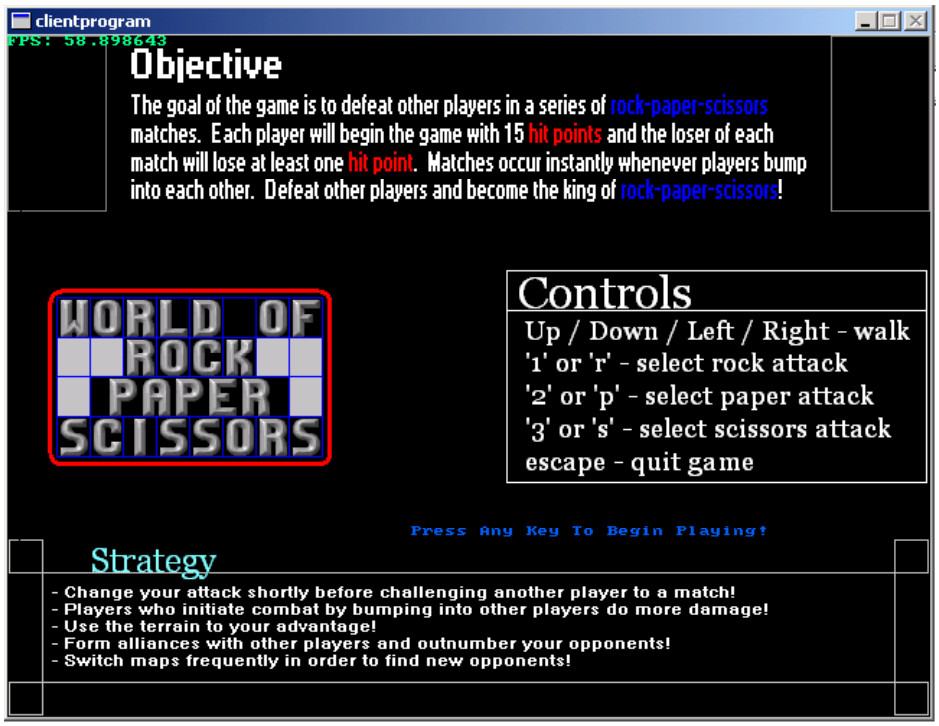


Figure 2: The title screen for *World of Rock, Paper, Scissors*

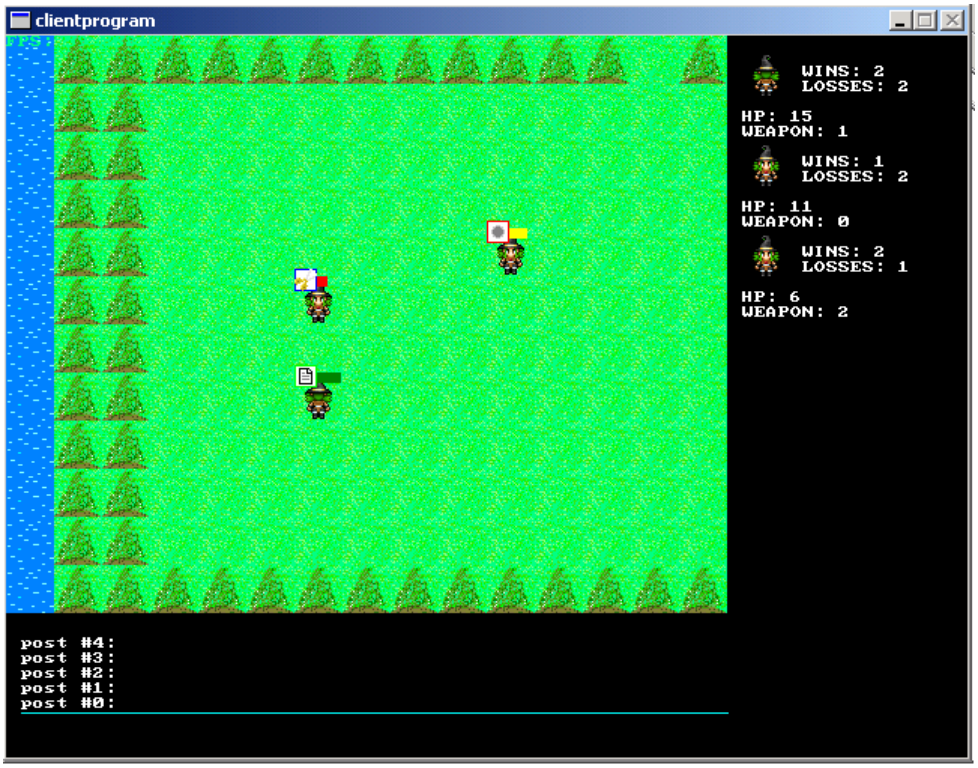


Figure 3: An in-game screenshot with three players facing-off.

3.2.2 The Program Model

Our implementation begins with the server program. The server program performs two tasks in an endless loop. The first task is that the server must listen for new clients who wish to log into the game. When one is found, the server then takes that client and assigns it a place in the game world, according to the rules. The server then adds this client to its growing (and shrinking) list of all currently connected clients allowing other players to have knowledge of the player from this client and must tell all other clients about the new client. The second action that the server must take is that it must accept and respond to every command that the each of the clients send. We implement this phase of the server's logic by using a for-loop over the entire list of clients and calling a special command interpreting function on each packet in each player's socket. (Each packet is a separate command. This will be explained shortly.) This loop is where the majority of the work is done in the server program.

The client program begins by reading a configuration file. This file contains the well-known IP address and port that the game server is running on. If the client cannot connect, or if the connection is refused, then the client terminates. After the client connects then it simply waits for updates from the server. While the client is waiting for updates, the player can take any of the actions that are specified in the rules. When these actions are taken they are sent to the server as commands. The normal client program also draws the game world and the player's avatar to the screen. The special versions of the client program that were created for the bots do not render an image to the screen. Since the majority of the work in the system is done by exchanging events and commands,

these will be defined in section 3.2.6. This section describes when each event is used and how both programs process it.

Our peer-to-peer messaging modification works by sending selected events directly from one peer to the other peers on the same map instead of sending them to the server first. In our game, a client's area of influence is the map they are on. Hence, whenever a player takes an action, the only players who need to receive notifications are the players on the same map. With our peer-messaging plan, each client has to know the IP address of its neighbors. This is accomplished by including each player's IP address with the rest of that player's data when it is first sent to a client upon joining a new map. For our experiments, we choose to send only the chat messages over the peer-to-peer network, as opposed to player state information - such as, hit points, wins, losses, and other information that could undesirably affect the game if intentionally modified by another player.

Our peer-content server modification works by having each client act like simulated members of the game's actual distributed server network. The client-server architecture of our system would allow security to be maintained by having each client verify everything (which could be a very simple and fast process as opposed to having the server send the content modification) that it obtains from another peer with the real server. (The implementation of this feature, however, is beyond the scope of our project and not actually implemented in our game). The program that would make this verification process possible is called the MapServer. This program is explained in the next section.

3.2.3 The MapServer Program

Massively multiplayer game servers frequently have to serve large updates. These can be either permanent alterations, like a patch, or they can be large temporary updates, such as the contents of a new zone when a player arrives at a new location. In both cases, the server is burdened with having to transmit large chunks of data. This data could be several kilobytes or several megabytes, depending on the situation and the game.

In our game, we require the clients to download maps that they haven't been to yet from the server. This is our way of simulating these larger updates. The maps, however, are not served by the actual server process. They are instead served by a separate process that we call the MapServer. There are several good reasons for implementing this as a separate program. First, by making this program a separate process, a distributed server can more easily be implemented. For example, a game might have one server program and two MapServers running on three different computers. Another reason for running the MapServer as a separate process is so that we can easily detach it from the server and allow clients to run it to decrease the number of tasks that the server performs.

With the implementation described above, the peer-layer is transparent. No matter where the MapServer is actually located, a client can connect to it and get a map without knowing if the update was from the server or another player.

In our game, the MapServer’s IP addresses are given out by the server. During the course of a game, a player may determine that it needs a certain map to continue playing. It is at this time that the server gives the client the IP address of a MapServer. This can be either the address of a client or of the server. The client then uses this IP address and the

well-known port, 25556, to obtain the update. If the MapServer is unable to meet the request for any reason, then the client just simply reissues the request to the server again.

3.2.4 Maps

The game world is made up of a 10x10 grid of maps. Each map is 15 tiles in width by 12 tiles in height. The maps are created using a map editor, which we created from the original concept demo of the game. An avatar is used as the cursor, which can be moved around the map like in the game (See Figure 4). The “J” and “K” keys cycle through the background (passable)/foreground (non-passable) tiles (each background tile has a corresponding foreground tile) and the “B” and “F” keys place background and foreground tiles respectively. “N” and “M” cycle through the map I.D. numbers. Each map has an I.D. between zero and ninety-nine. When a map is saved, the file name is the map’s I.D. number with a .map extension. Maps are saved to disk with the “W” key and old maps can be loaded with “L”.

When maps are saved, they are written to a binary file with the first thirty-two characters containing the file name and 180 characters (15x12) making up the map. The binary file type was chosen to prevent the world from being easily modified by the players.

Figure 4: The Map Editor allows maps to be created and modified with relative ease. The user simply moves the avatar onto tiles placing either background (i.e. grass, bridge) or foreground (i.e. tree, water) tiles.

3.2.5 Program Messages

In this section we detail the complete list of game events that are exchanged between the clients and the server. Some of these events are player-initiated commands and others are server-generated updates. A few of these updates are used in both ways. In our game, each event is sent in a separate packet. This way both the server and the client can process each packet independently of each other. In our source code and in this section the terms “packet”, “event”, and “command” are all interchangeable.

Connection Packet

First, when a new client needs to connect to the server it must begin the transaction by sending a connection packet to the server. The username and avatar fields should be set the way the player wants them to be. The client should fill in zero for the ID and one for the type unless the player is trying log back in as a previous player. Then the ID should be the ID from the session that this client wishes to resume. Now the server will respond with another packet. This packet will contain the player's new ID and the other two fields will contain the values that the client must use for their avatar and username. Usually, these are the requested values anyways. If the server returns an ID of 0 however this means that the connection was denied. This can happen for a number of reasons, including having a re-login attempt fail. (When the chosen ID is already in use for example).

Step Packet

Whenever a client wishes to take a step it must notify the server of this action by announcing only the direction of the step. This is because only the server knows the true position of the player if the player lags and becomes de-synchronized. (And, this is the only way for the server to handle conflict-resolution with stepping). The client can still animate the avatar and move the avatar but the client must correct any inconsistencies due to lag later when a position packet arrives from the server, if that is necessary.

Position Packet

First, a client must initiate a step using a step packet. When the server processes the step packet it will then send a position packet to update every client that needs to know this information. The management of the animation “parts” (fractions of a step) are merely cosmetic and are calculated by each client. Only the server sends this packet and the server only sends this packet in response to a step packet.

Player Packet

The server sends this message to a client whenever the client needs to add a player to their list of fellow players on the same map. This can happen because the client has recently joined a new map, or has recently joined the game, or because another player has joined the map. The fields in this packet specify every attribute about the new player, including the IP address for the peer-to-peer system to use.

Leave Packet

The server sends this message to a client whenever the client needs to remove a player from their list of current players. This occurs every time a player voluntarily leaves the map, when a player on the same map is defeated and moved elsewhere, and when players log off. This message is only sent by the server to the clients. Clients do not inform each other of their own leaving, for security reasons.

HP Packet

This packet is sent by the server to a client to notify that client that another player's hit points have changed. The fields in this packet contain the new value and the

player ID of the player who needs to have their value updated. Players must not be trusted to send this update themselves either.

Weapon Packet

First, a client will send this packet to the server with both ID fields set equal to that client's ID. The weapon field should be set to the weapon that the client wants to switch to using one of the enumerated constants representing rock, paper or scissors. After the server receives the update it will send one of these packets to every player on the same map as the original player. This is how the weapon information is updated. (This is virtually identical to the HP packet, except the player chooses the new value first).

Win-Loss Packet

The server sends this message to a client whenever the client needs to update the number of kills and deaths for a given player. Like the HP and Weapon update events, this event is restricted to the server.

Map Packet

When a client discovers that it needs a map it will send a request using this packet. The client will set the ID field to its own ID and set the mapid field to the mapid that it needs. Failures should be set to zero and the other two fields will be undefined. Now, the following steps happen:

1. The server replies with the address field filled out. This is a MapServer's address.

2. The client attempts to use this addr to get the map from that MapServer.

3. If the attempt fails for any reason, like connection broken, failures is incremented. Repeat from step 1.

4. If the attempt succeeded, then the client now has their missing map.

Chat Packet

When someone wants to send a chat message they must first copy their message into the message field. The player ID field can be set to specify a private message to one player. If the ID field is set to 0 then the message is said “out loud” and it will be sent to everyone on the same map as the speaker.

Ping Packet

If a client sends a ping packet to the server then the server will reply by returning the ping packet unchanged. However, the client can use this to detect the latency of the system. First, the client should sequentially label each of its ping requests using the serial field. This prevents confusing one ping’s start with another ping’s end. (That causes a much lower time than it should be). Second, the client should fill in the start field with any consistent timing metric that it chooses. An example of this would be milliseconds since the program began. That way, when a client hears a ping reply, the client will know how long the latency is.

3.2.6 Programs for the Experiments

Besides the normal server program, client program, and MapServer, four other programs were created specifically for the experiments.

First, our experimental design required the ability to run multiple players all simultaneously. To do this, we removed the graphical display from the client program and implemented four different bots. This changed program is called the BotProgram in the experimental design (Section 3.3). This new BotProgram can run up to 265 bots at once. (The limit of 265 was imposed by the number of sockets that we were able to open).

In addition to this, our implementation of the peer-to-peer chatting was hard coded into the program. As a result, we ended up with two different versions of the BotProgram: one with peer chatting enabled and one with that feature disabled. The latter program was used as a control in the experiments. Our server program also had two versions: one where the client-operated MapServers were used whenever possible and another version where the server program never redirected a map request to a client MapServer. And again, the latter program was needed as a control in the experiments.

The Experimental Bots

The four different bots were each created to test a different part of the system. Below are brief explanations of each bot and how they determine which actions to take:

ChatBot: This bot merely stands still and sends a random chat message every 400 milliseconds. The time interval of 400 milliseconds was chosen because this is the speed at which players normally take actions. For example, a player who holds down a directional key will take steps at the rate of one step every 400 milliseconds. This is

much more chatting that a human player could ever reasonably do. However, this amount of traffic allows us to test the effectiveness of sending small (96 bytes) but frequent messages over the peer network.

ExplorerBot: This bot walks in a straight line as fast possible. And when it hits a wall it always turns right. The purpose of this bot is to load as many maps as it can as fast as it can. This purpose is effective because ExplorerBots who bump into each other can pull each other out of areas where they are stuck. This bot was designed to test the MapServer.

AttackBot: This bot blindly moves toward the nearest player while using the shortest path. It also changes its weapon to be the weapon that will win against which ever weapon that player is using. AttackBots test the combat aspect of the program, and they are only used as a sub section of SpazBots in the experiments.

SpazBots: SpazBots earn their name from their behavior. The SpazBot is programmed to perform the actions of both the ChatBot and either an ExplorerBot or an AttackBot for each action depending on if there is another player on the map that the SpazBot is on. The purpose of a SpazBot is to take the maximum number of actions possible, and thus to load the server.

MeasurementBots: The purpose of this bot is to record the ping times from the server. This bot pings the server every fifteen seconds and saves the results until the program ends. When the program is over, the bot prints these times to the screen.

3.3 The Experiment Specifications

3.3.1 Experimental Data

Our experiments collected three different sets of data from the system. The three measurements that we will take will be the server's frame counter, the server's total bandwidth used, and the latency that is observed by a MeasurementBot.

The first measurement, the server's frame counter, requires an explanation. Our server program (all versions of it, refer back to section 3.2.5) keeps a count of how many times it completes its main loop. Recall that this main loop involves looping through each client once and processing each command from each client. Logically, the more clients that a server has, the longer each loop will take. Also, the longer each command takes to process, the longer this loop will take. And since an increased server load causes the main loop to run for longer, it will also execute fewer times. Therefore this measurement allows us to infer the server's relative processing load.

The total bandwidth used by the server is calculated from the number of packets sent by the server program and the number of maps served by the official MapServer. Since a single packet is always 96 bytes in length and a single map is always 213 bytes in length, the formula is bandwidth = maps*213 + packets*96. This measurement is necessary because we also want to minimize the amount of bandwidth that each client uses.

Bandwidth is an important constraint to consider when loading a server. Although most massively multiplayer game servers have access to high bandwidth lines, each individual server can only output so much data per second, no matter how much data it can process.

Our third measurement, latency, is the most commonly used metric by players to determine how loaded a server is. Our experiments measure the latency every 15 seconds so that we can record the latency spikes and the average latency of an entire game. The server's latency indirectly depends on how fast the server completes each frame. Therefore, we can use this measurement to infer total server load as well.

3.3.2 Experiment Design

To run the experiments we used three computers. All three of the computers were plugged into a private LAN using a hub. No other computers were present on the LAN and no other network traffic was present on the LAN while the experiments were being run. This way we could prevent interference and unnecessary packet collisions. We also tested this hardware with a single client to verify that the latency introduced by the hardware was less than one millisecond. Since all three computers were physically only a few feet apart, this was expected.

The standard procedure for running the peer-to-peer chat trials was as follows: the ServerProgram and MapServer were run on the server machine. BotPrograms, which are used to specify how many of a certain type of bots are to be run, are executed on the client. One BotProgram, running only a MeasurementBot, was executed on a laptop computer, while another, running all other bots, was executed on the client computer. Before the BotProgram is run, several parameters must be set. For the BotProgram that is

not running the MeasurementBot, the parameters set are the independent variables in these experiments, such as: game length (two minutes), bot quantity and type (75-150 chatBots or spazBots) and type of chat implementation (either peer-to-peer or server). At the end of each trial, the BotPrograms will print their statistics (those described in Section 3.3.1) to the screen.

The procedure for testing the client-side implementation of MapServer is similar to the peer-to-peer chat trials with a few differences. In these trials, a MapServer is also run on the client machine. The BotProgram parameters have also changed: game length can now be either two and four minutes, explorerBots are used in quantities of 100, 125, and 150, and MapServer location is the final independent variable which can either be just one server-operated MapServer or both a server and client-operated MapServer.

The final set of trials we ran was a combination of the previous two sets. The first four used server-implemented chat and MapServer, while the last four used client-implemented chat and MapServer. All experiments in this final set were two-minute games using 75, 100, 125, and 150 spazBots.

4 Results and Analysis

The following four sections of the report illustrate and analyze the results of our two experiments. Our first experiment tested the effectiveness of our peer-to-peer sub-layer against a similar system that did not have this feature. The results of those experiments are graphed in section 4.1 and the analysis of those graphs is in section 4.2. Our second experiment tested the effectiveness of allowing the clients to operate their

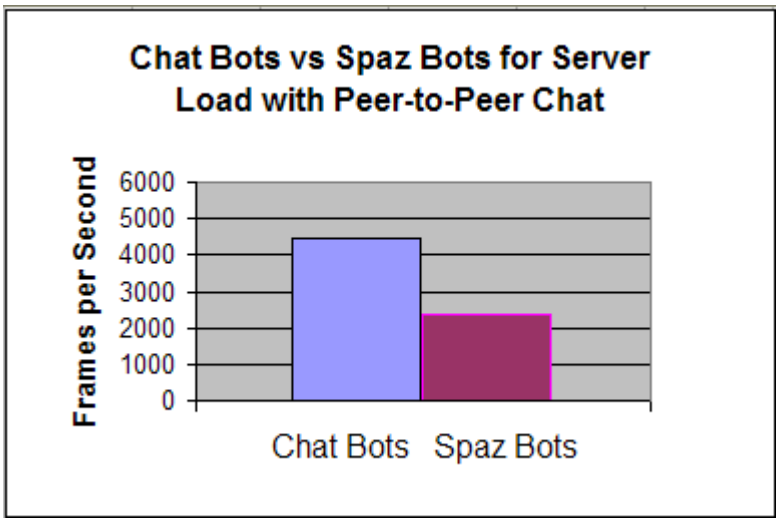
own MapServers versus only running the MapServer program on the server. The graphs of that data are found in section 4.3 and the analysis of those graphs is in section 4.4.

4.1 Results of the Peer-To-Peer Sub-layer Experiments

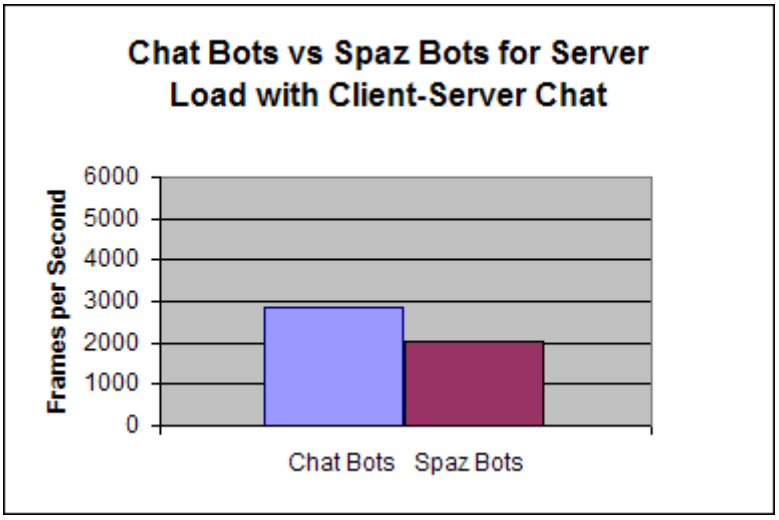
This section of the report shows the data that we obtained from the peer-to-peer sub-layer experiments in graph form. During our experiments we used three independent variables: the use of the peer-to-peer sub-layer (on or off), the type of bot used (a ChatBot or a SpazBot), and the number of bots that are connected to the game (75, 100, 125, or 150). However, our analysis only utilizes the results from the trials where 150 bots were used. Section 4.2 analyzes the data that we graphed in section 4.1. For the purpose of this analysis we chose to focus exclusively on the data where the experiments used 150 bots. We chose to analyze the data in this way because the data from the non-150 experiments was similar to the other data sets. Specifically, the number of bots in the game had an approximately linear effect on the server's frames-per-second and bandwidth used, and no predictable effect on the latency of the server between trials. However, our results for the 150-bot trials are consistent with the rest of the runs. The complete table of results can be found in the appendix.

Graph 4.1.1 compares the difference in frames-per-second that the server was able to achieve for both ChatBots and SpazBots while using the peer-to-peer chat system. Graph 4.1.2 shows the data from the similar games that only used the conventional client-server model instead. Our frames-per-second data was calculated by dividing number of logical frames that the server completed during the two-minute game by the number of seconds in the game, 120. The important difference between the two types of bots is that

the SpazBot sends chat messages while playing the game whereas the ChatBot only chats without moving.



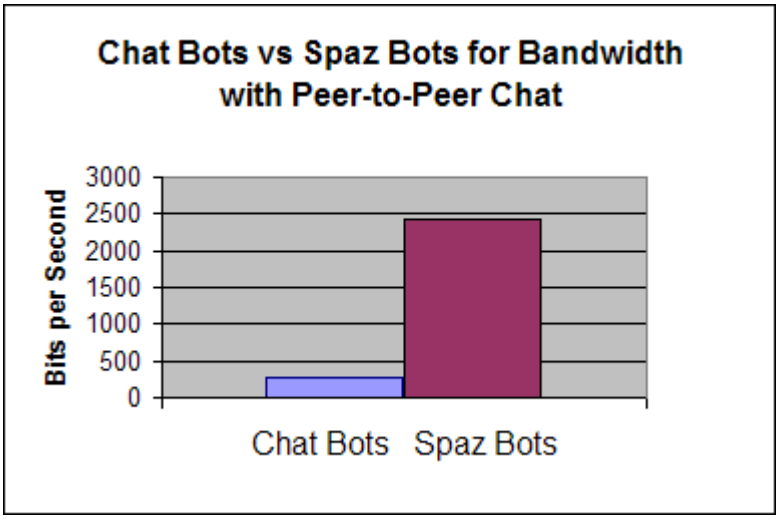
Graph 4.1.1: Chat Bots vs. Spaz Bots for Server Load with Peer-to-Peer Chat



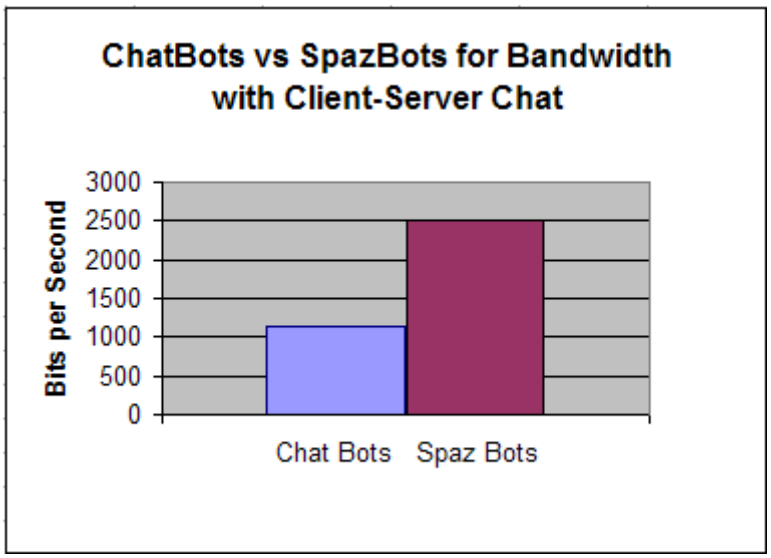
Graph 4.1.2: Chat Bots vs. Spaz Bots for Server Load with Client-Server Chat

Graphs 4.1.3 and 4.1.4 compare the difference in bandwidth (via total bits per second) used for both of the two bot types. Graph 4.1.3 shows the results from the peer-to-peer system and Graph 4.1.4 shows the results from the client-server system. Our total bits were calculated by the sum of every packet sent by the server and also every map served by the MapServer. Each packet is 96 bytes long and each map is 213 bytes long;

and the exact count of both was recorded for each game. These numbers can be found in the appendix. (The simple multiplication is not shown here.) To calculate the unit, bits per second, we divided the total bits by the game length, 120 seconds.



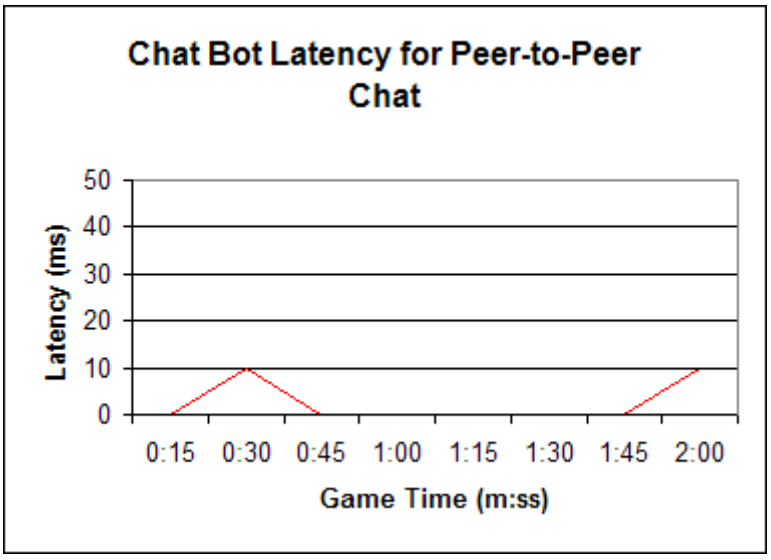
Graph 4.1.3: Chat Bots vs. Spaz Bots for Bandwidth with Peer-to-Peer Chat



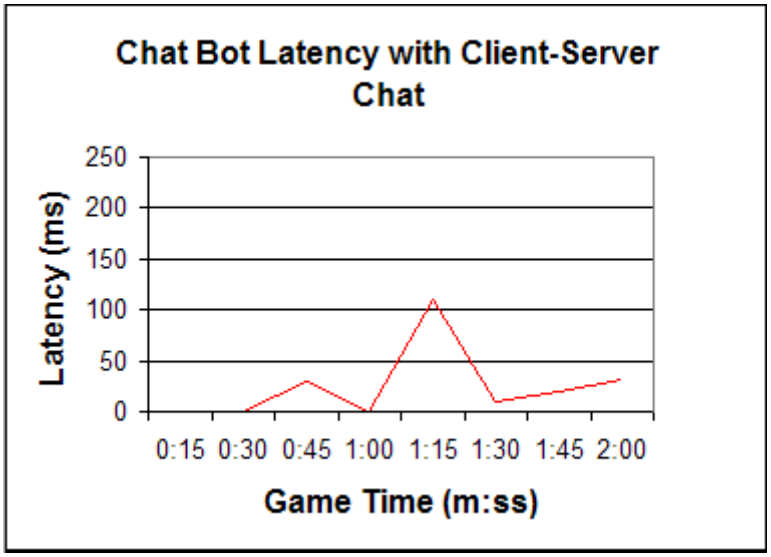
Graph 4.1.4: Chat Bots vs. Spaz Bots for Bandwidth with Client-Server Chat

Graphs 4.1.5 and 4.1.6 are line graphs that show the latency of the chat bots during games that used the peer-to-peer chat implementation and the client-server implementation of chat, respectively. One important thing to know about the observed latency is that latency under 100ms is undetectable by the player and latency under

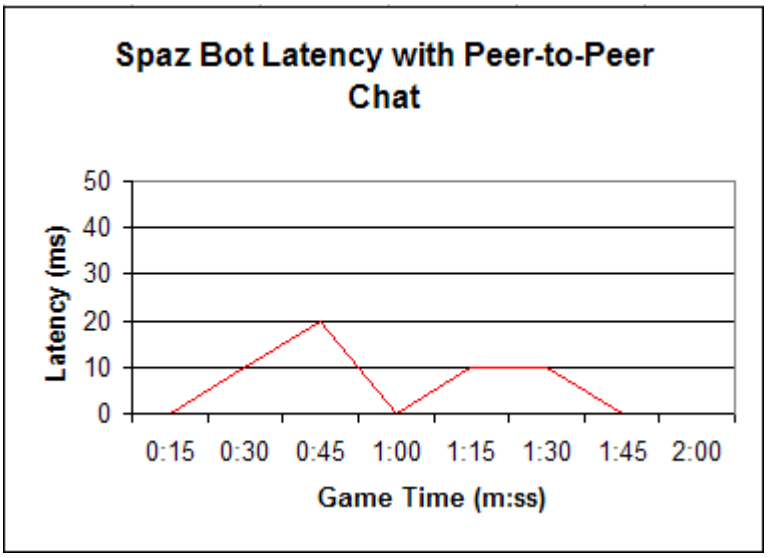
400ms is considered comfortable. These thresholds were determined by the client implementation: it only sends updates every 100ms and the player is limited to one real-time action every 400ms. Another important detail is that our latency measurements have a +10ms margin of error. This is due to timer granularity. Since we have no way of remedying this, we have to accept this margin of error.



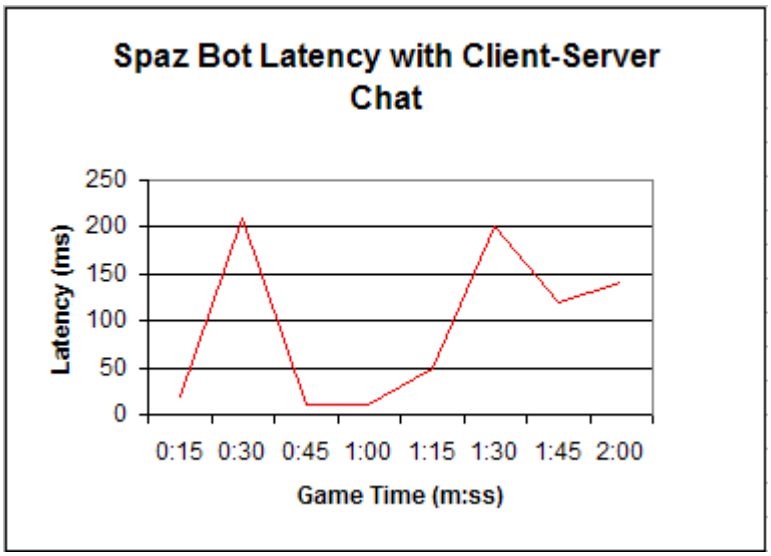
Graph 4.1.5: Chat Bot Latency with Peer-to-Peer Chat



Graph 4.1.6: Chat Bot Latency with Client-Server Chat



Graph 4.1.7: Spaz Bot Latency with Peer-to-Peer Chat



Graph 4.1.8: Spaz Bot Latency with Client-Server Chat

4.2 Analysis of the Peer-to-Peer Sub-layer Experiments

The first two graphs of this section show the effect that our peer-to-peer sub-layer has on the server's frames per second. With the peer-to-peer sub-layer, the server has a 66% improvement in frame rate when running with chat bots. With spaz bots however, there is very little change. This difference is due to how much more processing the spaz

bots require. With the spaz bots, the server spends the majority of its time processing their movement commands. Even though the spaz bots send just many chat messages just as frequently, the server processes them much faster. The chat bots however do not send any movement commands and 100% of their processing time is devoted to relaying these chat messages. Therefore, the chat bots show the largest improvement.

The peer-to-peer sub-layer also affects the bandwidth consumption of the server in a similar way. Once again, the chat bots show a large improvement while the spaz bots are not significantly affected. This difference is due to how many more updates a single step command requires. Normally, both a single chat message and a single step command result in exactly one update going to every player on the map. This update is either the new position of the player or the content of the text message. However, when players move between maps or when players attack each other, multiple updates must be sent to each client. These multiple updates greatly outnumber the quantity of chat messages that are sent. Also, spaz bots generate extra traffic by changing their weapon multiple times in between in each step and each chat message. Since each weapon change requires that an update be sent to everyone, this single action is guaranteed to generate more traffic than their chat messages. Once again, since chat bots only send chat messages, they show the largest improvement here.

Comparing graphs 4.1.5 and 4.1.6 against graphs 4.1.7 and 4.1.8 show that the latency of the system is lower in two ways when peer-to-peer messaging is used. First, the highest latency spikes are drastically lower when peer-to-peer messaging is used. (Compare 10ms

and 20ms against 111ms and 201ms). Second, the average latency is also lower for the peer-to-peer sub-layer enabled games. This difference exists even for the spaz bots that were unaffected in the other two metrics. The reason that the spaz bots were affected here is because the spaz bots are just as prone to the server lagging as the chat bots are. In other words, the server is a bottleneck. And when the traffic arrives in bursts, like in this experiment, there will be times when even the most capable server will not be able to keep up. So it is possible that peer-to-peer messaging can help alleviate bursting traffic.

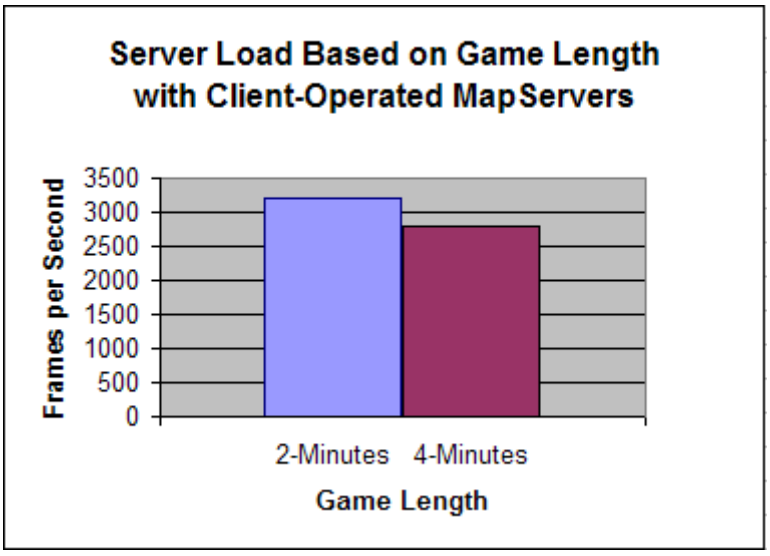
Since the chat bots benefit the most from peer-to-peer messaging, this means that the most appropriate messages to select for use on a peer-to-peer sub-layer would be the ones that are the most frequently sent. And if a game has any traffic that tends to come in bursts then that traffic might be a good candidate as well if it is not game critical traffic that must be sent to the server first. Our analysis of the spaz bots show that the usefulness of a peer-to-peer sub-layer is negated when a majority of the traffic is not sent over this network.

4.3 Results of the Client-Operated MapServer Experiments

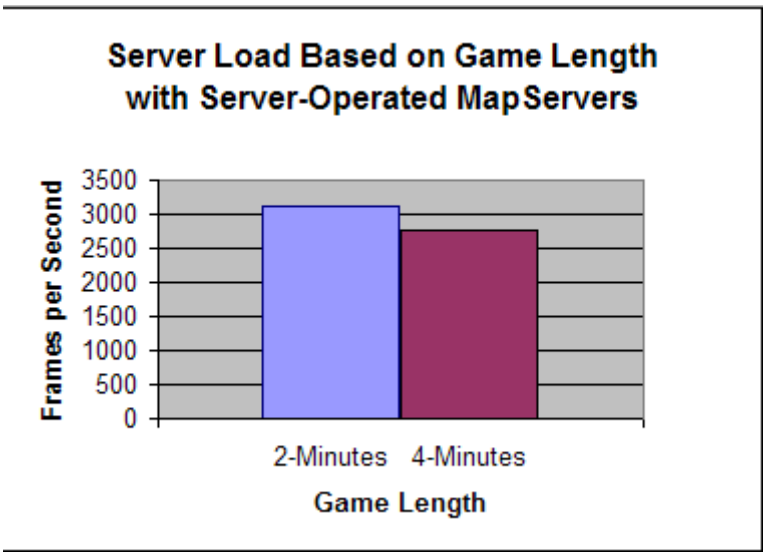
This section of the report shows the data that we obtained from the client-operated MapServer experiments in graph form. During our experiments we used three independent variables: if clients were allowed to host their own MapServers (yes or no), the length of the game (2 or 4 minutes), and the number of explorer bots in the game (100, 125, or 150). However, our analysis only utilizes the results from the trials where 150 bots were used. Our reason for only analyzing part of the data is explained in section

4.4. This section of the report contains only the graphs that are referred to in the analysis. The complete table of results can be found in the appendix.

The first and second graphs of this section compare the frames-per-second achieved by the server during two minute and four minute games. Graph 4.3.1 shows the results of a system that is running client-operated MapServers. Graph 4.3.2 shows the results of a system that where each client must download each map from the server.

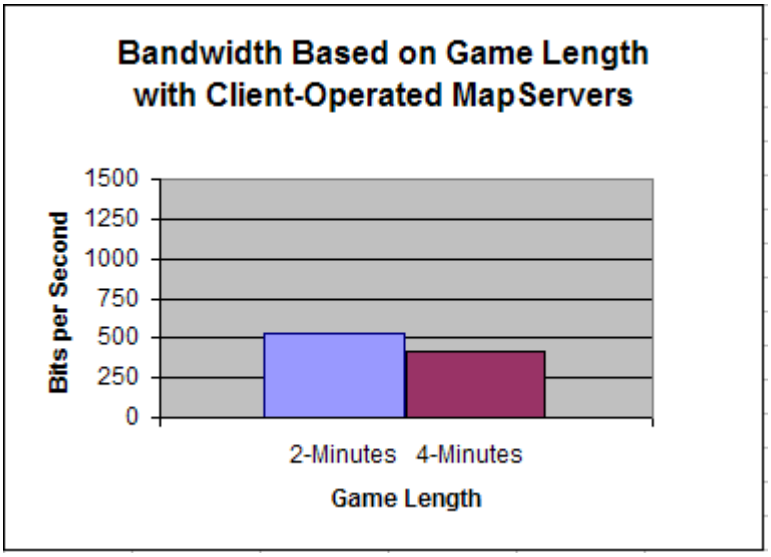


Graph 4.3.1: Server Load Based on Game Length with Client-Operated MapServers

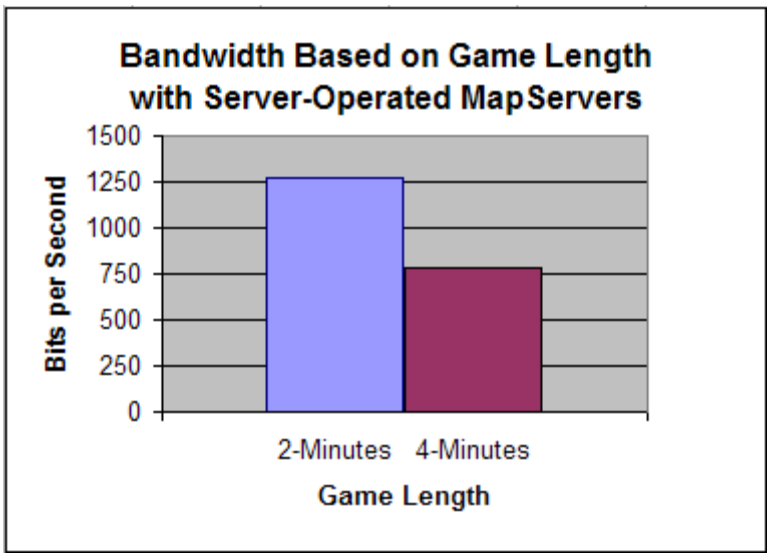


Graph 4.3.2: Server Load Based on Game Length with Server-Operated MapServers

The third and fourth graphs of this section show the bandwidth usage of the servers in the two and four minute games. Graph 4.3.3 shows the bits-per-second of a server that is aided by client-operated MapServers. Graph 4.3.4 shows the bits-per-second of a server that is serving every map by itself.

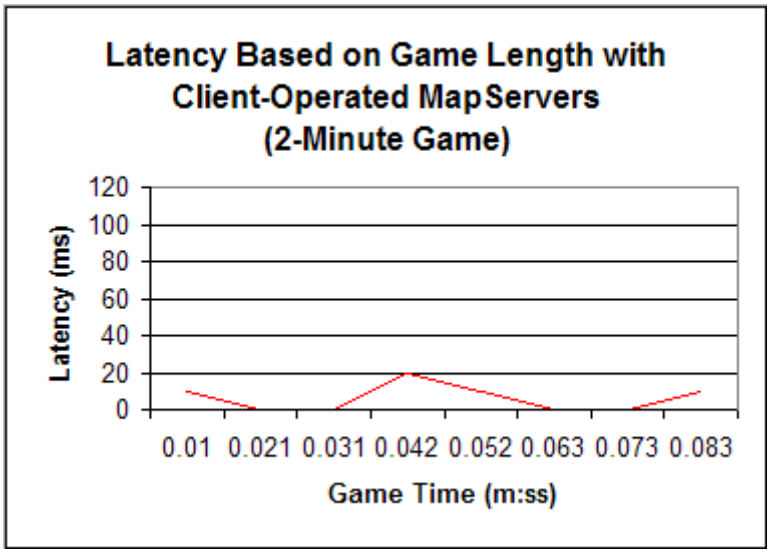


Graph 4.3.3: Bandwidth Based on Game Length with Client-Operated MapServers

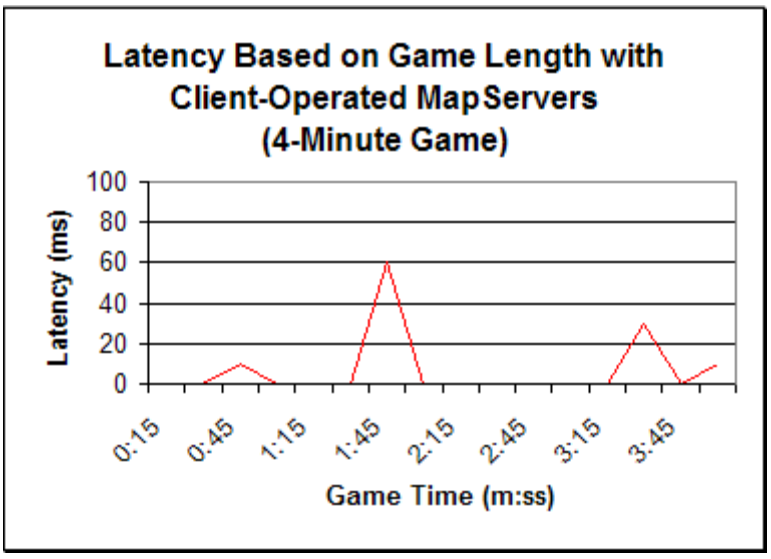


Graph 4.3.4: Bandwidth Based on Game Length with Server-Operated MapServers

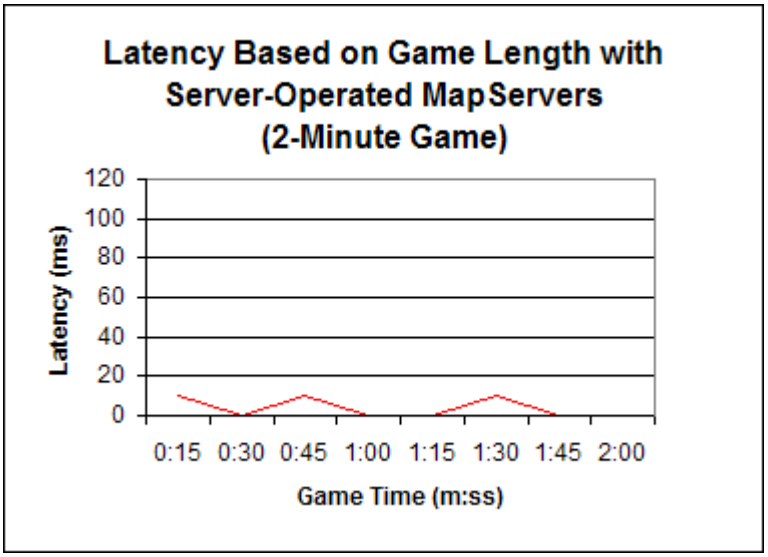
The next two line graphs show the latency observed by the explorer bots in two games that use client-operated MapServers. The last two line graphs show the latency in similar games that only use the one server-operated MapServer.



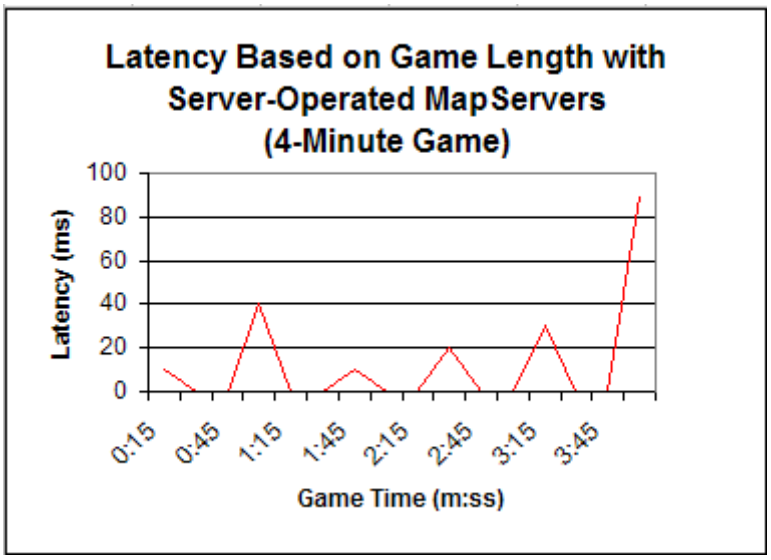
Graph 4.3.5: Latency vs. Game Length with Client-Operated MapServers (2 minutes)



Graph 4.3.6: Latency vs. Game Length with Client-Operated MapServers (4 minutes)



Graph 4.3.7: Latency vs. Game Length with Server-Operated MapServers (2 minutes)



Graph 4.3.8: Latency vs. Game Length with Server-Operated MapServers (4 minutes)

4.4 Analysis of the Client-Operated MapServer Experiments

The first two graphs of section 3.3 show almost no difference in server frames-per-second while client-operated MapServers are both enabled and disabled. This interesting result is caused by the fact that maps require very little processing to serve. In order to serve a map, all that the MapServer has to do is listen for a request and reply by promptly sending the map and closing the connection. So essentially, running a MapServer is not a processor intensive application.

Graphs 4.3.3 and 4.3.4 however do show a very significant difference between a system that uses client-operated MapServers and one that does not. In graph 3.3.3, the system that uses client-operated MapServers saves approximately half of its total bandwidth! This is a remarkable savings considering that each map is only downloaded once while the game messages are constantly sent while the program is running. What this means that is even though the maps are only 2.21 times the size of a normal update, they still account for half of the total bandwidth. However, notice that the bandwidth gains

decrease over time. This is because in our experiment each bot only needs to download each map once. And fewer maps are downloaded in the last half of each game than in the first half of each game.

Graphs 4.3.5 and 4.3.7 have lower latency on average than graphs 4.3.6 and 4.3.8. The difference between these two sets of graphs is the game length. Although the data seems to indicate that 2-minute games have less latency than 4-minute games, this does not make sense as a difference. Even though the largest latency spikes occurred at 2 minutes and 4 minutes exactly in the two 4 minute games, the placement of the latency spikes does not make sense. If anything, the latency spikes should have happened earlier in the game when more of the maps were being downloaded. So instead we have to conclude that our latency data for these experiments was more likely just variance in latency and making the difference not statistically important. If the single largest latency spikes in each game are ignored, then the average latency for each game is under 20ms. With this analysis we can say that the game length should affect the latency. This analysis does make sense since we observed no noticeable change in server frames-per-second earlier.

From our analysis, running client-operated MapServers clearly has the largest effect on the bandwidth of the system. Systems that use client-operated servers should use them to send data that is only significantly larger than the smaller gameplay updates.

5 Conclusions and Recommendations

The primary goal of our project was to find techniques to increase scalability in client-server architectures. We developed two such techniques that can reduce server load and, thus, allow for more scalable systems. Sections 5.1 and 5.2 discuss the benefits and what contributions can be brought to the domain of massively multiplayer games through the use of peer-to-peer messaging and peer content servers, respectively. In Section 5.3, we talk about possible modifications and enhancements can be made to this project for future MQP groups or those interested in Game Development.

5.1 Recommendations for Uses of Peer-to-Peer Event Messaging

Peer-to-Peer Messaging (P2PM) had the greatest effect on server load and a minimal effect on the total amount of bandwidth used. For those reasons, P2PM works best for small, player-initiated events that happen frequently. However, this may not apply to every game. Consequently, P2PM may not be useful for every game. Recall that during the experiments, both the chat bots and the spaz bots were programmed to send a chat message every 400 milliseconds. This is an unrealistically fast rate for a typed conversation to take place at. In the real world, if our 150 spaz bots were replaced by 150 real players, then the server load would have to be lighter. This would lessen the effect that P2PM has on the system, too. We would have liked to have tested this, but organizing 150 computers and 150 human participants was prohibitive. Our implementation of the server program and the client program would allow us to conduct this experiment because we used bots.

Another situation where P2PM is useful is when messages between two players must be exchanged so rapidly that keeping the server in the loop is too expensive of an operation. This happens when the acceptable latency of the current operation drops below

the current average latency. In The World of Rock, Paper, Scissors, the acceptable latency is 400 milliseconds and the point at which latency is not even noticeable is around 100 milliseconds. This is because the player's two fastest moves require a 400 and 100 millisecond delay before they can be preformed again. (These actions are taking a step and changing one's weapon, respectively). Since the player is being forced to wait for a period of time that is greater than the latency, the latency does not matter. But now imagine an even more fast-paced version of the same game where the delays are lowered to 100 milliseconds and 50 milliseconds, respectively. Now that the players can more easily reach the latency limits of the system, they may find it more acceptable to handle 'duels' on their own machines and then update the server one second later.

Although we only used chat data in our peer-to-peer messages in the experiments, this does not have to be the only implementation of this idea. In a real implementation, this data could be any non-gameplay data that is not time sensitive. Chat traffic is just one example of this. Another example could be if our game allowed players to change their avatar image at their preference during the game. Then that data could be sent to a player's peers first and to the server later. Another example would be putting each client in charge of informing its peers of that player's win-loss record. In that case, the server would follow up the peer's message with the 'official' message whenever the server had the resources to do so. And if there was a significant discrepancy between the two updates then that could be investigated. (Usually, the suspicious message will turn out to be either extremely outdated or corrupted. But a tamper-resistant packet-ordering scheme, perhaps also with checksums, would let each client and the server detect when duplicate messages and intentionally altered messages were sent). Small and frequently

occurring messages are the best candidates for P2PM. The messages that are sent over a P2PM network may be non-critical or critical, or somewhere in between.

5.2 Recommendations for Uses of Peer Content Servers

In our experiments we used map data to test the effectiveness of having client-operated content servers. Although this is a good use for such a system in a real game, there are more uses. For example, if the geography of the world is included in the game's installer, then the updates to those maps could be the map data that is sent. Modern massively multiplayer games have much more happening in them than just the geography of the world and the players running around on it. Other game objects include server-controlled agents (such as enemies) and resources to collect. It may also be possible for players to affect the geography of the world itself, or it may change over time. All of these things could mean that players would have to keep downloading updated game-world parameters often. This places a significantly larger burden on the server than simply handling the game events, as our analysis shows.

Another potential use of client-operated content servers could be using them to serve program updates and patches to their peers. These types of mandatory updates pose a greater problem to a server, not because of the size of the download or the number of times that it must be downloaded, but because of the demand for it. If a software patch is mandatory then the currently active players will all need it immediately. Other active players who login the same day will also need the patch immediately. So the situation is that a sizeable portion of your total userbase will all demand the same update all at once.

This makes serving the patch difficult. If the decision is made to have a dedicated patch server then this server will be idle most of the time every day and it will be overloaded and unable to meet the demand of the players the few times that it is needed. Having enough additional patch servers to accommodate for the demand is even more wasteful than having just one server because then more servers become idle. So a good solution would be to have the one dedicated patch server, but then to have the clients download the patch directly from each other when the patch server is busy. (The hardware and bandwidth expenses for this patch server could even be saved by taking one of the game servers offline to temporarily 'seed' the client network with the patch.) Blizzard Entertainment has recently implemented a system even more advanced than this in their recent game *World Of Warcraft*. This new patch delivery system involves having groups of clients share the downloaded patch in a BitTorrent-like fashion (Blizzard Entertainment – Inside Blizzard: Legal FAQ). (BitTorrent works by having peers ‘share’ a download by using their combined upload bandwidth to upload different parts of the same file to each other. These parts are then recombined by the client when every part is acquired. BitTorrent uses cryptographic hashing (SHA1) of all data and Blizzard likely takes no less care as well.)

The best use of a client-operated content server is to relieve the game server in a large, single update that will go to many clients. Our analysis shows that the bandwidth savings would almost certainly be large and meet the needs of the game. The data that the client-operated content server is serving does not need to be constant, like a patch, either. If a player is playing the game primarily in a certain area, then they will probably have

the most recently updated version of that area in their client. This client could then 'tell' another client who has just entered the area everything that it knows. Of course, clients should also have to verify knowledge gained in this way to be secure. But then that new client could begin telling other clients what it knows, and so on. In this way the game server could send a minimal amount of updates to more players.

Our results also indicated a significant improvement of latency when client-operated content servers were used. However, this was because we were running the map server and the game server simultaneously on the same physical machine. In a real MMG implementation there would have to be many different machines running in parallel to handle the load. So, the latency caused by having the combined servers is a non-issue. Nonetheless, if one's peers are chosen intelligently then an improvement in latency could still be possible. For example, two players in Massachusetts that directly connect to each other have less latency versus the same two players bouncing messages off of the nationwide east-coast game server in Florida.

5.3 Future Work

Because the goal of our project was not so much to create a game, but rather to enhance the scalability of the Client-Server model, there are a number of modifications that can be made to our project perhaps for a future MQP in either Computer Science or Interactive Media and Game Development. First is Voice Over Internet Protocol (VoIP), which is a voice chat scheme that is used in some of today's MMG's, such as *Phantasy Star Online*. This method of chat is like text-based chat in that it is non-critical data and

can be handled via P2P. Another modification is the use of a distributed server farm, which would, instead of having the game run off of a single server machine like we did, have several dedicated server machines, thus enabling load distribution across multiple machines. The last possible modification would be the use of a load balance detection algorithm in conjunction with the distributed server farm to detect when a certain machine gets overloaded. When this happens, this piece of code can move future clients to another server.

6 Works Cited

1) Yahn W. Bernier. “Latency Compensating Methods in Client/Server In-game Protocol Design and Optimization” Valve Software. March 2001
2) Takuji Iimura, Hiroaki Hazeyama, Youki Kadobayashi. “Zoned Federation of Game Servers: a Peer-to-Peer Approach to Scalable Multi-player Online Games.” SIGCOMM’04 *Workshops*, Aug. 30+Sept. 3, 2004, Portland, Oregon, USA. P116-120. ACM 1-58113-942-X/04/0008
3) Chris Gauthier Dickey, Daniel Zappala, Virginia Lo. “A Fully Distributed Architecture for Massively Multiplayer Online Games”. SIGCOMM’04 *Workshops*, Aug. 30+Sept. 3, 2004, Portland, Oregon, USA. P171. ACM 1-58113-942-X/04/0008
4) Nick Feamster, Hari Balakrishnan, Jennifer Rexford, Aman Shaikh, Jacobus van der Merwe. “The Case for Separating Routing from Routers.” SIGCOMM’04 *Workshops*, Aug. 30+Sept. 3, 2004, Portland, Oregon, USA. ACM 1-58113-942-X/04/0008
5) Shun-Yun Hu, Guan-Ming Liao. “Salable Peer-to-Peer Networked Virtual Environment” SIGCOMM’04 *Workshops*, Aug. 30+Sept. 3, 2004, Portland, Oregon, USA. P129-133. ACM 1-58113-942-X/04/0008
6) Blizzard Entertainment – Inside Blizzard: Legal FAQ. *http://www.blizzard.com/legalfaq.html*