2006-03-09

# Design and Measurement of a Real-Time Peer-to-Peer Game

Michael D. Simonsen
*Brigham Young University - Provo*

# DESIGN AND MEASUREMENT OF A REAL-TIME PEER-TO-PEER GAME

by

Michael Simonsen

A thesis submitted to the faculty of

Brigham Young University

in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computer Science

Brigham Young University

March 2006

BRIGHAM YOUNG UNIVERSITY

GRADUATE COMMITTEE APPROVAL

of a thesis submitted by

Michael Simonsen

This thesis has been read by each member of the following graduate committee and by majority vote has been found to be satisfactory.

| | |
|---|---|
| Date | Daniel Zappala, Chair |
| Date | Dan Olsen |
| Date | Christophe Giraud-Carrier |

BRIGHAM YOUNG UNIVERSITY

As chair of the candidate's graduate committee, I have read the thesis of Michael Simonsen in its final form and have found that (1) its format, citations, and bibliographical style are consistent and acceptable and fulfill university and department style requirements; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the graduate committee and is ready for submission to the university library.

_____ _____

Date

Daniel M. A. Zappala
Chair, Graduate Committee

Accepted for the Department _____

Parris K. Egbert
Graduate Coordinator

Accepted for the College _____

Thomas W. Sederberg
Associate Dean, College of Physical and Mathematical Sciences

# ABSTRACT

DESIGN AND MEASUREMENT OF A REAL-TIME PEER-TO-PEER GAME

Michael Simonsen

Department of Computer Science

Master of Science

Currently, multiplayer online games use the client-server architecture which is very resource intensive, expensive, and time consuming. Peer-to-peer protocols are a less resource intensive alternative to the client-server model. We implement a peer-to-peer protocol called NEO in a multiplayer game and run experiments in a lab setting and over the Internet. These experiments show us that NEO is able to run a smooth playable game, with low unused updates and low location error. This happens as long as the arrival delay is long enough to allow updates to arrive in the given time limit and the round length is short enough to keep the location error down. However, the experiments also show that NEO has scalability problems that need to be corrected. When more than 4 clients are used the playout delay is the same length as the round which causes high location error. Also, more clients cause more updates to go unused which also causes high location error.

## Contents


**1 Introduction** **1**

1.1 Current Practice . . . . . 1

1.2 NEO . . . . . 3

1.3 Thesis Summary . . . . . 3

**2 Related Work** **5**

2.1 NEO . . . . . 5

2.2 Other Approaches . . . . . 8

2.2.1 Mercury . . . . . 8

2.2.2 Synchronized P2P simulation . . . . . 8

2.2.3 SimMUD . . . . . 9

2.2.4 Sync-MS . . . . . 9

2.2.5 Booster Boxes . . . . . 10

**3 Implementation of a Peer-to-Peer game using NEO** **11**

3.1 Building an Online Multiplayer Game . . . . . 12

3.1.1 Eater Game . . . . . 12

3.1.2 The Game Loop . . . . . 14

3.1.3 Dead Reckoning . . . . . 16

3.2 Client Server . . . . . 18

3.3 NEO . . . . . 21

3.3.1 Positional vs. State Changing Updates . . . . . 21

3.3.2 Sending Updates . . . . . 23

3.3.3 Storing Updates . . . . . 24

3.3.4 Vote Processing and Counting . . . . . 24

3.3.5 Processing Updates . . . . . 25

3.3.6 Time Synchronization . . . . . 27

**4 Experimental Setup** **29**

4.1 Data Collection . . . . . 29

4.2 Lab Experiments . . . . . 31
4.3 Distributed Experiments . . . . . 32

**5 Results** **34**
5.1 Lab Experiments . . . . . 36
5.1.1 Short Arrival Delays . . . . . 36
5.1.2 Increasing Round Length and Pipeline Depth . . . . . 40
5.1.3 Best Performance . . . . . 41
5.1.4 Client-Server Comparison . . . . . 42
5.1.5 Scalability . . . . . 43
5.2 Distributed Experiments . . . . . 49
5.2.1 Short Arrival Delays . . . . . 49
5.2.2 Increasing Round Length and Pipeline Depth . . . . . 51
5.2.3 Best Performance . . . . . 55
5.2.4 Scalability . . . . . 56

**6 Conclusions** **61**
6.1 Protocol Issues . . . . . 62
6.2 Potential Improvements . . . . . 64

**7 Appendices** **66**
7.1 The Mono Project . . . . . 66

# 1 Introduction

Video games have become one of the most popular uses for the home computer. The video game industry reported sales of 7.3 billion dollars in 2004 [1], more than double the sales in 1996. Among the people who are playing video games, 43% play some sort of online game. These multi-player online games have become very popular among internet users. Multi-player games range from turn-based games [2], which are not very demanding of resources, to real-time strategy (RTS) games, which need at least one move every 500 ms, and then to first-person role-playing (FPRP) games that require a latency of under 100 ms to be playable [3]. The resource requirements for these games have caused game providers to create large server farms with large amounts of bandwidth and processing power. These server farms are necessary to keep the games running and meeting their latency requirements. The growth of this industry is expected to continue, so there is a need to find other ways to allow these games to run without having to use more and more hardware.

One way to support large-scale online games without a large number of centralized resources is through the use of peer-to-peer networking. In order to use a peer-to-peer protocol for a real-time game, the protocol will need to meet latency requirements, be secure, and be scalable. This is a difficult thing to do because it is hard to find the right balance between security and latency. Generally the more security there is in the protocol, the longer it will take and the higher the latency will be.

## 1.1 Current Practice

Currently most multi-player games use a client-server model for communications. In this architecture, a client sends an update to the server, which then verifies the contents of the update and sends it to all other clients that need the information. This architecture is very simple to implement and keep running because the server is responsible for computations, data storage, ordering events, and security of all these items, but this can also be quite a drain on the resources of the server.

One of the main limitations that multi-player games have is the network. The amount of bandwidth a game needs directly affects the number of clients that can connect to the server at the same time. The amount of bandwidth needed for a FPRP server is shown in Pellegrino et al. [4] to scale quadratically with the number of clients; while in most games the bandwidth requirements do not rise this dramatically, they are much higher than linear. Companies do the best they can to provide high-speed connections. However, bandwidth can still be constrained by how much is available in the geographical area in which the servers are located. While the bandwidth requirement at the client does scale linearly, the slower speeds of connections at the client make it so that the client is still limited in the number of updates it can receive.

Latency is another issue that multi-player games have to deal with in a client-server architecture. The high bandwidth use can lead to queuing delays on routers near the server, causing large and jittery latencies. Having to send all information through a server also can cause high latencies for clients. As discussed earlier, different kinds of games have different tolerances for different amount of latencies. First-person games need a very small latency because the state of the game can change very rapidly; the longer it is between updates the harder it is to make an informed decision in the game. A RTS game can handle a much higher latency because this type of game does not require the same level of control as a FPRP does. As long as the amount of jitter is low, RTS games can have latencies of up to 500 milliseconds without affecting the game [3]. Jitter is a problem because users like a game to update at constant intervals. If the jitter is high, users feel like the game is jerky, which can be very frustrating for the users.

The main way that companies help these games to scale is to have server farms. These server farms are not the same kind of server farms that are used for websites. In a website server farm, all the servers are serving the same information and any one of the servers can process any given request. Multi-player games use different kinds of server farms. One kind has each server running its own instance of the game; there is no interaction between the servers. In a distributed game server farm, each server runs a portion of the game; a player is transported from server to server when they

move to a new area in the game [5, 6]. In addition, multiple server farms are usually set up for a single game and each server farm is separate from the others, so there is no communication between them. Both of these types of server farms help with the scalability problems, but they are still limited to just a few thousand clients. In July of 2002 Everquest had more than 400,000 active users [7]. Limiting these players to only be able to interact with a few thousand other users takes away a large portion of the game experience.

## 1.2 NEO

Peer-to-peer networks allow clients to communicate with each other directly, instead of sending the communication through a server. This can help to alleviate some of the strain placed on the server by real-time games. NEO [8] is an example of a peer-to-peer architecture that was built for real-time communications. One of NEO's primary accomplishments is security, as it prevents several common methods of cheating. However, the performance of NEO in a real world environment has not previously been studied.

In this thesis our goal is to design and implement a game using the NEO protocol. We are doing this because NEO has never been implemented in a real game; it has only been simulated. Designing a game with NEO will help flush out issues that may not have come up during the simulations. We are measuring NEO's performance so that we can determine how well it works in a real game over the Internet. The goal is not to scale NEO to $400,000$ clients or even $2,000$ that is used in a game like Everquest. Instead the goal is to get NEO to scale to groups of $10 - 50$ clients without using a central server. This thesis is limited to looking at the communication and not the computation or storage aspects of NEO.

## 1.3 Thesis Summary

For this thesis, we design, implement, and measure the performance of a real-time, peer-to-peer game using NEO for multiplayer communication. We conduct our measurement study in both a lab environment with emulated delay and loss, as well as

on the Internet. In both cases we use artificial intelligence to play the game remotely. We examine many parameters to determine the feasibility of using a peer-to-peer architecture for real-time games.

One of the contributions of this thesis is the modification of NEO to distinguish between state-changing and positional updates. This modification is done by using a TCP channel for state-changing moves and the regular UDP NEO channel for positional updates. This allows NEO to not have to recover any lost update, because an update that is lost will be out of date by the time it is recovered. It also guarantees that the state-changing moves will arrive at the clients, this keeps the game state on each client synchronized. Another contribution this thesis makes is NEO settings that allow the game to be played effectively with $3-4$ players. These settings allow the game to be played with low location error which makes the game more playable. Obviously, if NEO can only scale to 3 or 4 players it will not be very effective as a multiplayer protocol, so we find the problems that keep NEO from scaling and point out ways to solve these problems.

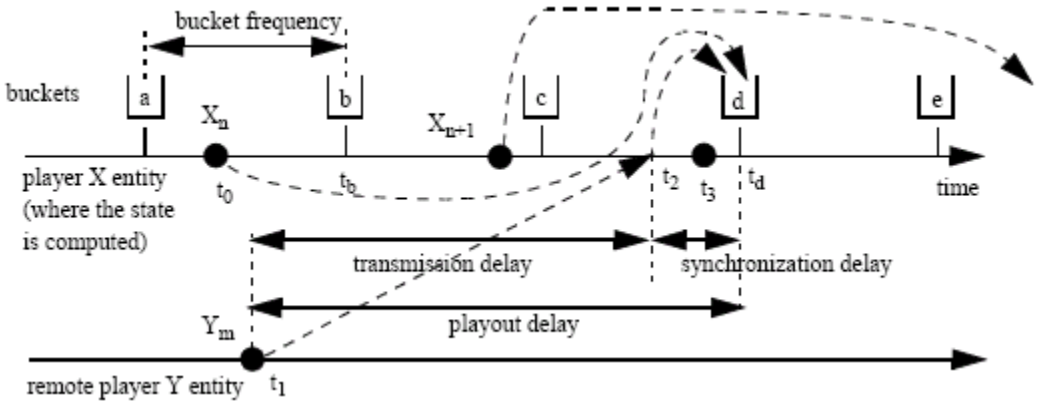


Figure 1: Bucket synchronization mechanism used in MiMaze. The horizontal is time [10]

## 2 Related Work

Many ideas have been researched and tested which try to make peer-to-peer games possible. We begin by examining the development of NEO, then consider alternative approaches.

### 2.1 NEO

Any distributed multiplayer game needs to use a method called *dead reckoning* to help overcome problems with latency [9]. Dead reckoning is a method of guessing where the opponents will be at time $t$ based on where they were at time $t-1$ and $t-2$. This allows the user to react more quickly to other clients and makes for smoother game play. It can also make wrong guesses, which cause the clients to jump around the screen unrealistically, so using as little dead reckoning as possible is best for a game.

One of the first distributed real-time games was MiMaze [10]. MiMaze uses multicast as its main means of communication, and a system called bucket synchronization as a way to keep the game state consistent. In bucket synchronization, shown in Figure 1, time is broken down into periods called buckets. All events that happen

during one of these time periods are used to compute the new game state after a given amount of time. As seen in Figure 1, all events that happen between times $a$ and $b$ are put into bucket $d$ and used to compute the state of the game at that time. These updates are placed in bucket $d$ because this is the bucket which occurs after the transmission delay time, so each update has enough time to get to its destination. In this way all updates from other clients have time to arrive so they can be used to compute the game state.

Bucket synchronization has two fundamental limitations. First, it is bounded by the latency of the slowest participant because the time before a bucket is processed needs to be long enough for the moves to arrive. Second, bucket synchronization is also susceptible to cheating. A client can simply not send out messages for a period of time in order to see what is going on in the game and then send out a message with an update. Other clients will simply think that the malicious client has a poor connection and that the update is consistent with what it was going to do all along.

The lockstep protocol [11] attempts to fix the security problems in bucket synchronization. In this protocol each client decides its next move but does not send it to the other clients. Instead it sends out a one-way hash of the move; once a client receives hashes from all other players the client sends out the plain text of its move. This makes it easy for each client to verify the moves of the other clients. It also makes it so that it is not beneficial to wait before an update is sent out. The main problem is that the game play is still slowed down to the speed of the slowest participant.

Some modifications have been proposed for the lockstep protocol, such as the pipelined lockstep protocol [12]. This pipeline modification allows the protocol to send the hash of a certain number of updates before the actual updates are sent out. This allows the lockstep protocol to not be affected as much by the speed of the slowest participant but it also introduces a suppressed update cheat. This means that a malicious client can choose to not send an update until it has more information than it should for the update it is sending.

The adaptive pipeline lockstep protocol [13] was proposed to help overcome

the problems of the pipelined lockstep protocol. In this protocol the value of the latency of the slowest link is used to determine the size of the pipeline to use. This protocol helps to eliminate the problems with the pipelined lockstep protocol, but the writers acknowledge that it is possible for this algorithm to think a client is cheating when they are really just having temporary congestion on their link.

NEO [8] is another protocol that has been proposed to allow real time secure communications in a peer-to-peer system. NEO is much like the lockstep protocol except for one major difference: it is not bounded by the slowest client's latency. This is because NEO divides time into rounds. The round length is then the bound for the latency of updates. If an update is not received by enough clients within the round window it is not used to update the game state. This makes it so that each peer does not have to have good latency to every other peer, just good latency to enough peers so that its updates get acknowledged. Each update the peers send contains information about whether or not the peer got an update from the other peers for the last round. This voting system allows each peer to know if a move was accepted by enough peers to be used.

Several optimizations improve NEO's performance. Players can modify the NEO round length if messages keep coming in late or all the messages are getting in well before the time limit. This is done by another voting procedure; once the majority of nodes agree that the round length should be changed the new round length is set and broadcasted to all nodes. Rounds can also be pipelined like in the pipelined lockstep protocol. This gives NEO the same benefits that pipelined lockstep has, but it does not expose NEO to the same cheating because of the round length waiting limit.

A recent extension to NEO adds congestion control to avoid overwhelming the network [14]. Clients observe packet loss and then adjust the pipeline depth up or down to increase or decrease the sending rate, again using a group voting system to decide what to do. When changing the pipeline depth, NEO uses additive-increase and multiplicative-decrease like that used in TCP. NEO also uses an exponentially weighted moving average when determining when to change the pipeline depth so

that it is not affected by sudden temporary peaks in congestion.

## 2.2 Other Approaches

### 2.2.1 Mercury

Mercury [15] is a peer-to-peer architecture that takes the reverse approach of NEO. While NEO takes a bottom-up approach, making the architecture secure first then dealing with scalability, Mercury is scalable but not as secure.

Mercury uses a publish-subscribe architecture that can be implemented on any distributed hash table(DHT) such as pastry [16], chord [17], or CAN [18]. A query language similar to SQL is used to publish and subscribe to events. In a Mercury game a client that is in an area will publish events such as its position. When client 1 publishes its location, all other clients who are subscribed to the area that client 1 is in will receive an update message letting them know client 1's new location. This publish-subscribe system can be used for any type of object such as treasures or weapons, not just locations.

The performance evaluation of Mercury shows that the delivery delay scales linearly with the number of players. For 50 players the delay was an average of 400 ms and for 100 players the delay was an average of 500 ms. There are a few things that can be done in the DHT to make the delay less, such as taking into account the distances between nodes.

### 2.2.2 Synchronized P2P simulation

When the multiplayer portion of Age of Empires was made, the main goal was to keep the single player game play intact while supporting up to 8 players. This meant being able to support hundreds if not thousands of units on screen at once. Because of bandwidth limitations, it is not possible to send information about each unit to each player in the game. Instead each client runs its own simulation of the game and sends the players input, such as mouse movement and key presses, to all other clients. The clients then just have to make sure the simulations are

synchronized. If the game becomes unsynchronized the simulation stops and must be restarted by the players.

While this is a good solution for allowing thousands of units to be created and used at once it does not scale very well which is one reason the game only supports 8 players. Also the simulations can only run as fast as the slowest client can process updates and render the new game world.

### 2.2.3 SimMUD

SimMUD [19] is a peer-to-peer architecture that is built on Pastry and Scribe [20]. In SimMUD players are grouped together based on their location in the game space. These groups then share updates using multicast. Because a DHT and multicast are used the system has a high tolerance for loss and failure of nodes.

The game state is kept by nodes that are participating in the DHT network, with a server being used for game states that are not changed very often. Game state is also replicated onto other nodes that are not in charge of the state in case the node that is in charge of the state fails. When this happens the replica node takes over and the game continues without the players noticing.

SimMUD has an average latency of 200 ms which is good enough to play many types of multiplayer games but not all. Also if too many clients are in the same group the multicast will start to slow down causing game play to lag.

### 2.2.4 Sync-MS

There is another approach to peer-to-peer gaming being proposed by some researchers which does not include a new protocol. Instead they want to add hardware to the architecture of the internet which would make peer-to-peer gaming easier, one such proposal is call Sync-MS [21].

Sync-MS adds nodes the network that synchronize game play. These nodes only attempt to make the game fair, they do not attempt to lower delay. The nodes make the game fair by delivering the message at the same time to all other nodes and sorting out the order of player actions.

### 2.2.5 Booster Boxes

Booster Boxes [22] are extra servers that are deployed throughout the network which act the same way web caches work. They cache messages for faster responses, they aggregate messages so less messages have to be delivered, they filter messages so that only clients that need the messages get them, and they route messages so delay is minimized.

This allows a lot of the load to be removed from the server and distributed around the network while not putting the load on the clients. This also keeps the network more secure than a regular peer-to-peer network would be.

## 3 Implementation of a Peer-to-Peer game using NEO

The first thing we did was implement NEO with both a peer-to-peer and client-server network architecture. We did this so that game performance, using statistics such as latency, bandwidth, and playout delay, could be compared between the two types of architectures. Doing this involved creating an efficient game loop and dead reckoning algorithm. We also had to create the client-server architecture and work out the details of how to implement NEO for a real game.

## 3.1 Building an Online Multiplayer Game

### 3.1.1 Eater Game

We implemented a Pacman-like game that was called the Eater Game. This game, shown in Figure 2, uses Windows Forms and consists of a randomly generated maze and a few randomly placed stones. A new stone is randomly regenerated each time one is eaten so that there will always be a consistent number of stones on the board. The code base for this game comes from www.c-sharpcorner.com [23],but it was modified considerably to make it more suitable for the experiments. The object model and the way rendering was handled were modified to make them more efficient. Also multi-player support and network communication were added.

The maze for this game is generated by first creating a number of cells that cover the game board. Then starting with the cell in the top left corner a neighbor to that cell which has all walls still intact is randomly chosen. The wall between the two cells is then knocked down and the new cell is placed on the stack. This process is then repeated starting from the cell that was chosen. If the starting cell does not have any neighbors that still have all walls, a cell is popped off the stack and it is used as the starting cell. This process is continued until all cells have been visited. The algorithm then randomly chooses a number of cells and knocks down a few more walls. This is done to make the maze easier to navigate through.

The game is played by artificial agents using a modified A* algorithm. The first thing that the algorithm does is determine which are the five closest stones, ignoring walls. Next from those five it determines which is the closest, taking walls into consideration. If the closest stone is significantly closer than the other stones, the agent goes for the closest stone. This is done so that a agent will not turn away from a stone that is right next to its location. If it is not significantly closer, the agent randomly chooses a stone out of the five and uses the A* algorithm to determine the best path to take to get to the stone. The random picking from the top five stones is done so that if two agents are in the same cell they will not follow each other around the whole game. The agent is moved at a rate of 3 pixels every 50 milliseconds.

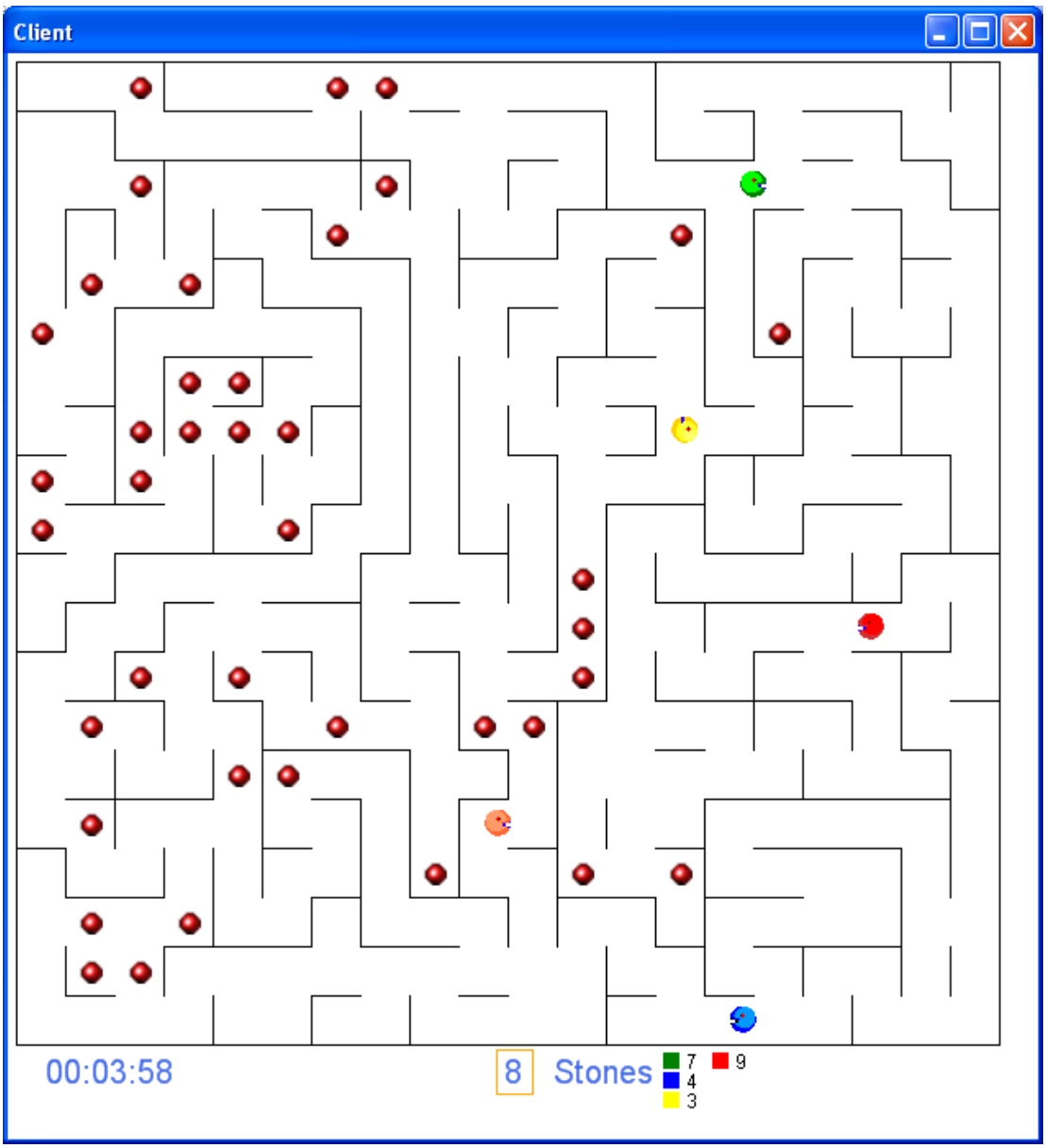


Figure 2: A screenshot of the Eater game

The goal of this game is simply to get the most stones in a specified amount of time, for our experiments this time was two to five minutes. The goal of the experiments is to observe the network communication and game performance, so while how many stones were eaten and who won the game is tracked while the game is played, these statistics were not saved for consideration in the results.

### 3.1.2 The Game Loop

The game loop is a very important part of building an efficient game. The game loop is what handles the order of processing events and updating the game state and rendering the game to the screen. A basic game loop looks like this:

**Algorithm 1** A Basic Game Loop

```
while gameIsRunning do
    Process User Inputs
    Update Game State
    Render to screen
end while
```

This algorithm has been shown to be the most efficient way of processing game state and updating the graphics for games. This is because there is one process doing all the work so there will be no fighting for processor time with other processes. If there are multiple agents on the screen at one time the process is able to update the information from all of them and then update the screen once. This is best because the updating of the screen is usually the most time consuming operation for the algorithm. If each agent instead uses it's own process to update the clients state and call for the screen to be redrawn, the redrawing will happen much more often and just a few agents will quickly overwhelm the computer, causing the game to run very slowly.

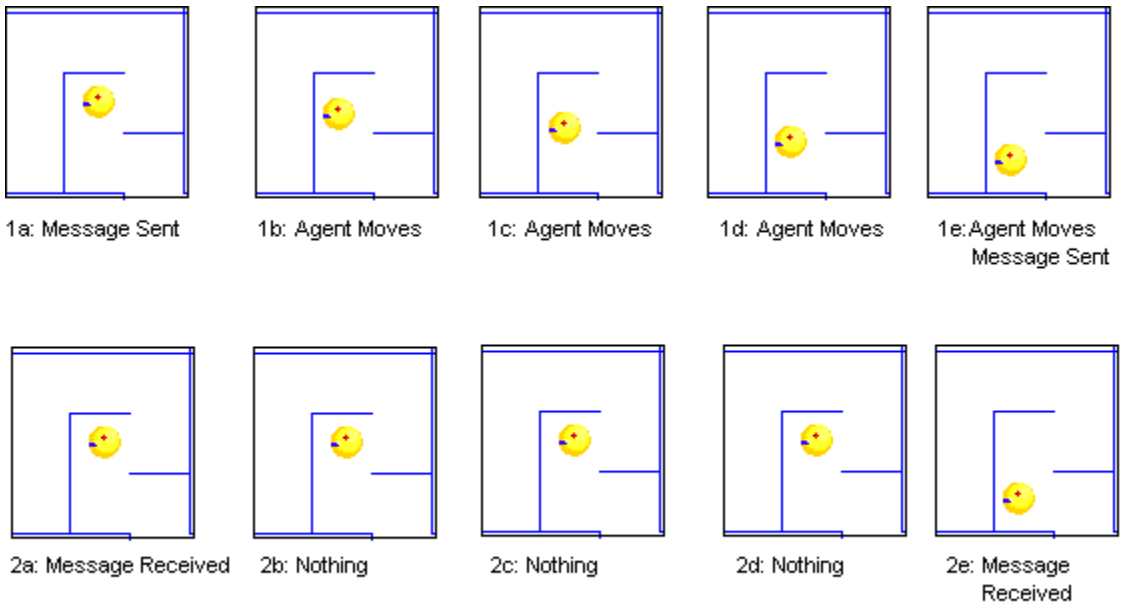


Figure 3: Example of how dead reckoning can help game play. The top line of images shows what is seen on the local client. The bottom line is what is seen on a remote client when dead reckoning is not used. If dead reckoning is used the top and the bottom images will be the same.

### 3.1.3 Dead Reckoning

One of the most important parts of a multiplayer network game is dead reckoning. As shown in Figure 3, dead reckoning allows each player to see the same game state at the same time. Basic dead reckoning is done by assuming that the distance and direction that an agent will move from time $t-1$ to time $t$ is the same as the distance and direction that the agent moved from $t-2$ to $t-1$.

We tried several methods to ensure good quality dead reckoning in the game. First, we used the basic dead reckoning approach which works well for games with agents like race cars that have to accelerate, but in a maze game like Pacman, in which the agents can stop and turn without any notice, this type of dead reckoning is not sufficient because it makes the agents appear to go through walls.

Our second attempt at dead reckoning in a maze was done using the A* algorithm to tell the dead reckoning algorithm which direction to go. This was not a good approach because a human user or even an AI algorithm with randomness will not always select the same stone that the A* algorithm does. This causes a lot of wrong guesses and ends up making the game look bad.

The third attempt was a combination of the first two. The A* algorithm is only consulted when the agent is in the middle of a square. This did not work because when the agent was in the middle of a square the A* might guess that the agent would go up, but instead the agent goes to the right. In the next square the A* guesses that the agent will turn around in order to go back to the square it had originally guessed, but again the agent moves to the right. So this hybrid method did not guess right very often and when it was wrong it would continue to be wrong for a number of guesses.

The method that was finally used was also a hybrid method but it has more conditions on when the A* is used. In this method the basic dead reckoning method is used unless the agent is in the center of a square and the agent's current direction will take the agent into a wall. Otherwise the A* method is used to see which direction that agent should turn. From our view this method turns out to work quite well in the limitations of a maze, because it keeps the agent from running into walls and

guesses pretty well when it needs to.

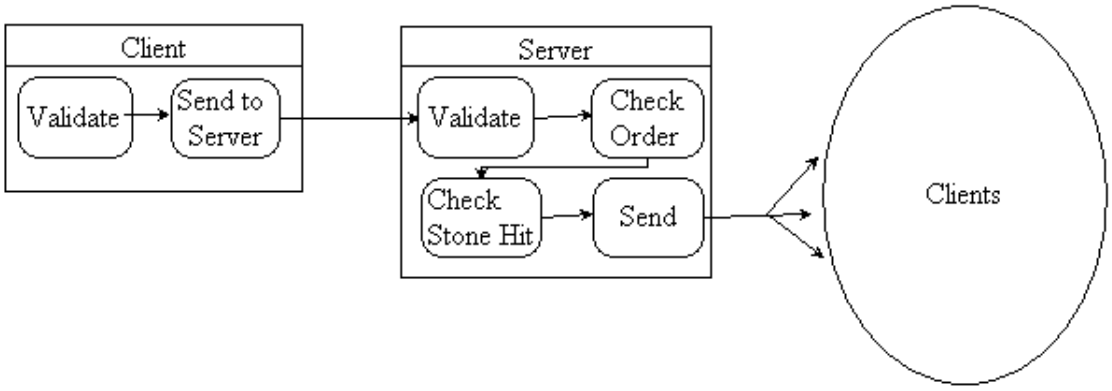


Figure 4: Update processing in the client server architecture.

### 3.2 Client Server

We implemented a client-server network interface so that its performance can be compared with the peer-to-peer architecture. Most online games use this model because the server is a central authority for the game. This makes it easier to manage the game because security, event ordering, data storage, and non-player characters are all controlled in the same central location. The second reason is that the updates can be processed at the client very quickly. All the client has to do is process the updates with the exact information that the server has sent to the client. This allows the client to spend very little time processing the messages and more time accepting user input and updating the screen. All messages in the client-server architecture are sent using UDP so that the comparison with NEO is more accurate; NEO uses UDP.

The simple client-server architecture we built is shown in Figure 4. The client only does a quick check to make sure the move is valid before it sends the update to the server; a valid move is one that doesn't go through a wall. When the server receives an update, it verifies that the update is valid and that the update has arrived in the right order. The server then sends the update to all agents except for the one it received the update from.

When the server checks an update, it looks for collisions with stones and walls.

An update that causes the agent to run into a wall is discarded. If an update causes the agent to run into a stone the stone is considered to have been picked up by the agent. When this happens, the server has to send a remove stone message to every client that is participating in the game. This is done by piggybacking the message on the update message that is already being sent to the clients. When there is a stone message going out with the update, the update is sent to all clients so that the client who sent the update will get a confirmation that it did pick up the stone.

When the update is checked, the server also makes sure that the update is not an old update, this is done by looking at the sequence number. Updates can arrive out of order because we are using UDP, if the update is late or out of order, the update is checked to see if the client picked up a stone and if that stone has not already been picked up by another client. If there is a need to send stone information out to the clients the server sends that information, otherwise the update information is discarded.

The way threads are used in the game is very important; if there are too few threads then the client will not be able to get every thing done in a timely manner. If the client uses too many threads, a race condition will develop, with one or more threads getting little to no processing time, which makes the process not able to get everything done that needs to be done. In the Eater game, the client has separate threads for the game loop thread, the network listen thread, and the update process thread.

The game loop thread is the main processing loop in the game. This loop is responsible for doing the calculations to determine where the agent should move. It then moves the agent and sends the update information to the server, it then updates the screen with the positions for all agents. This thread is also responsible for sending the statistics to the server at the end of each game.

The network listen thread listens for incoming updates and puts the new update on a queue for the update process thread to use. When there are updates on the queue, the update process thread pops one update off the queue on a first in first out basis. This update is then processed and the game state is updated with the new

information. Finally, the update process thread saves the statistics used for the game trace.

The server uses three types of threads: the stats thread, the server thread, and the update processing threads. The stats thread listens for statistics information coming in from the clients and compiles this statistics information. When the game is over it saves the statistics to the hard drive so it can be analyzed later.

The server thread listens for messages from the clients and puts them on a queue for processing and sending on to other clients; there is one queue for each client playing the game. There is also one update processing thread for each client; this allows the server to keep track of sequence numbers easier and it allows the server to process stone pick ups easier. The update processing threads pop messages off their queue on a first in first out basis. Then the thread processes the update information checking for valid updates and looking for stone pickups and sends the update information to the other clients.

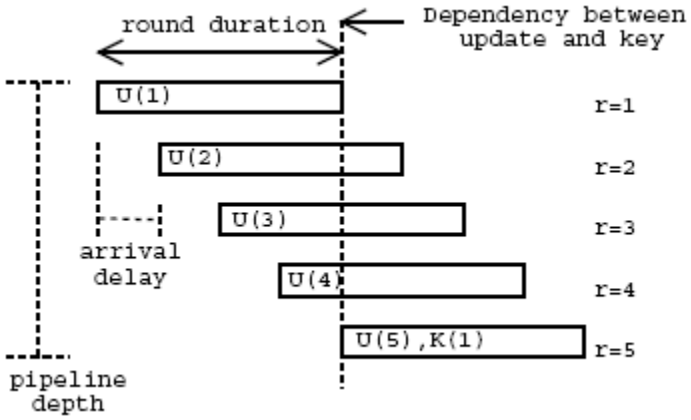


Figure 5: NEO rounds and pipeline depth description [8].

### 3.3 NEO

NEO is a peer-to-peer architecture that has been developed to allow real-time games to be played while not allowing cheating. Shown in Figure 5, NEO divides time up into rounds and adds a pipelined depth. NEO sends one update via UDP per round and depth. We added a TCP channel to NEO so that state changing moves could be sent with a guarantee of arriving and so that lost updates would not have to be resent over the UDP channel.

#### 3.3.1 Positional vs. State Changing Updates

When designing NEO into our game we decided that no messages would be recovered. This choice was made because the number of messages that would need to be recovered would be so much lower than the number of messages that could end up being recovered. Our method uses voting only for message acceptance, if a message is on time it is voted for and if a message receives at least half the votes it could get it is accepted otherwise it is discarded.

In order to make sure important messages still got through a server was used to send the stone update messages. This server communicated with the clients using TCP to make sure no updates would be lost. Since this these updates are a very

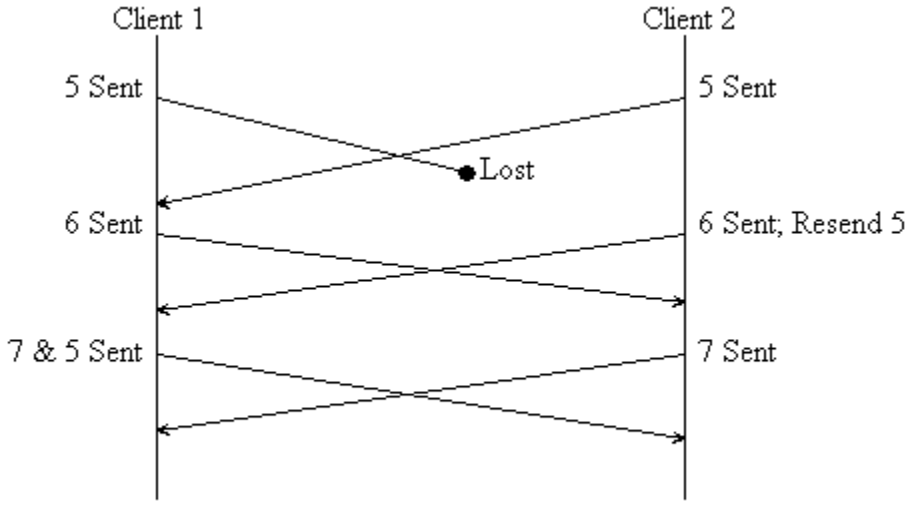


Figure 6: Message recovery situation. It can be seen that when client 1 resends update 5 it arrives too late to be useful, because update 6 has already arrived.

small portion of the overall update it has very little affect on the NEO system except that it allows us not to have to recover messages.

NEO uses a message recovery system that allows it to always have the information it needs for the voting to work. This is done by including a resend update flag in a the next update. When a client finds this flag in an update the last update will be automatically resent to the requesting client. This system requires all lost messages to be recovered so that the voting can be accurate. However, recovering some messages can take more effort than it is worth because most messages will be out of date by the time they arrive at the client and so the message will be thrown out anyway. An example of this in the Eater game is shown in Figure 6, since update number 5 gets lost a resend flag is set in update number 6, this causes client 1 to resend update 5. When update 5 arrives at client 2 update 6 has already been processed so processing update 5 would only cause the eater to jump backwards and then forwards.

Since not all messages need to be recovered in order to allow the game to run

correctly a system was devised to allow only the important messages be recovered. In the case of the Eater game the important messages are those which contain information about stone placement or removal. This system put a flag in the messages letting the clients know that the message was needed and should be recovered. This flag was to be put in the actual message and in the voting messages. This way if a message was lost the client could still tell if the message should be recovered. A problem occurs if two messages are lost, because the recover flag would be lost also. So the choice was do we add the recovery flag to the next n messages, n being a number that we are confident would not get lost in a row, or do we recover all messages. Since we did not want to recover all lost messages and recovering only certain messages was proving to be very complicated, we decided to go with the separate TCP channel which was described earlier.

### 3.3.2 Sending Updates

There are several specifics of the NEO protocol that had to be worked out so that NEO could be taken from theory to an implementation. The first is making sure that a move is sent out right at the beginning of the round so that it is sure to have the best chance of arriving on time at the other clients. The original method used to accomplish this was to queue up the next move during the preceding move. This method was good at getting the update to the other clients as quickly as possible but it was not a good method because the updates were more out of date than they had to be, moves could be as high as 1000 ms apart and the agents state is updated every 50 ms.

The method which was finally used was to split the sending of updates out to its own thread. This made it so that one thread was used to process AI input and update the game state, while another thread was used to send the agents position at the required time. This worked quite well because the sending of the updates was the only thing the thread had to do, so it was able to send the update at the very start of the round allowing the update to have the best chance of reaching the other clients within the time limit.

### 3.3.3 Storing Updates

The second thing that we added and designed for NEO was a system for storing updates. As specified by its designers NEO must keep all updates because they may need to be resent at any time. Since our modified NEO does not allow updates to be resent the only reason to keep the updates is so they can be sent in plain text with the next update. Even with the original version of the protocol a game will probably not need to keep all the updates that it has sent. With this in mind we developed an algorithm that keeps updates for a specified number of rounds before they are discarded. In our case the number was set at five to be sure that the updates were not discarded too soon. In the original protocol this number should probably be set to a much higher number, like 100, and a system would have to be added to the protocol which would inform a client that the update it is requesting has been discarded.

### 3.3.4 Vote Processing and Counting

Another item that needed to be created was the way voting was taken care of. How many votes update needs to be accepted was the main issue involved in voting; in the case of the Eater game this was set at 50%. The next thing that needed to be taken care of was how to store and retrieve votes efficiently so that the voting system does not cause lag on the update processing.

Using a threshold of 50% for the amount of votes needed causes an interesting side effect, when using 4 clients or less each update received by client A within the time limit will be accepted no matter how the other clients vote. This is because when an update is sent by client B and received by client A it is assumed that client B votes for it because it is the one who sent it and as long as it is received on time client A votes for the update. These two votes are the 50% needed for the update to be used. This situation makes the voting system break down for 4 clients and under, because when an update is received at client A it automatically receives 2 votes, one for the sender and one for the receiver, which is enough to pass the vote threshold. We see that in this case client B would only have to send a message to client A and

the message would be accepted even if client B did not send the message to the other two clients. One way to fix this situation is to not count a vote from the client that the update was sent from. This would make it so that a client needs to know that at least one other client has received the update for it to be accepted. For the purposes of this experiment it was decided that allowing a vote from the sending client was reasonable because the main goal was to test the protocol's speed and not its security.

The storing and retrieving of votes is done by checking an updates for votes as soon as it comes in. When an update has votes on it for other updates the votes are immediately recorded to the update they belong to, which is being stored for to be used in the round it belongs to. This allows for easy retrieving of votes because when an update is being processed it will have the votes with it making the votes easy to count.

### 3.3.5 Processing Updates

When and how updates are processed are two other items that needed to be examined. It was decided that updates should be processed if one of two things happens. The first thing that can trigger the processing of updates is if the round the update is part of is over. The second thing is if all the updates for a round have been received there is no reason to wait for the end of the round so the updates are processed.

How updates are processed is a more complicated process. Each client is running one thread for each other client which is in charge of accepting updates from another client. When an update is received from a client it is placed on a stack which contains messages for that round. The messages stay on this stack until one of the two conditions mentioned above is met. At this point the messages are popped off of the stack and placed on a processing stack, there is one processing stack for each client the updates are coming from. When a message is pushed on the processing stack it is a signal to the processing thread to pop it off and begin processing the update.

The first thing that is done shown on line 1 of Algorithm 2 is to make sure the

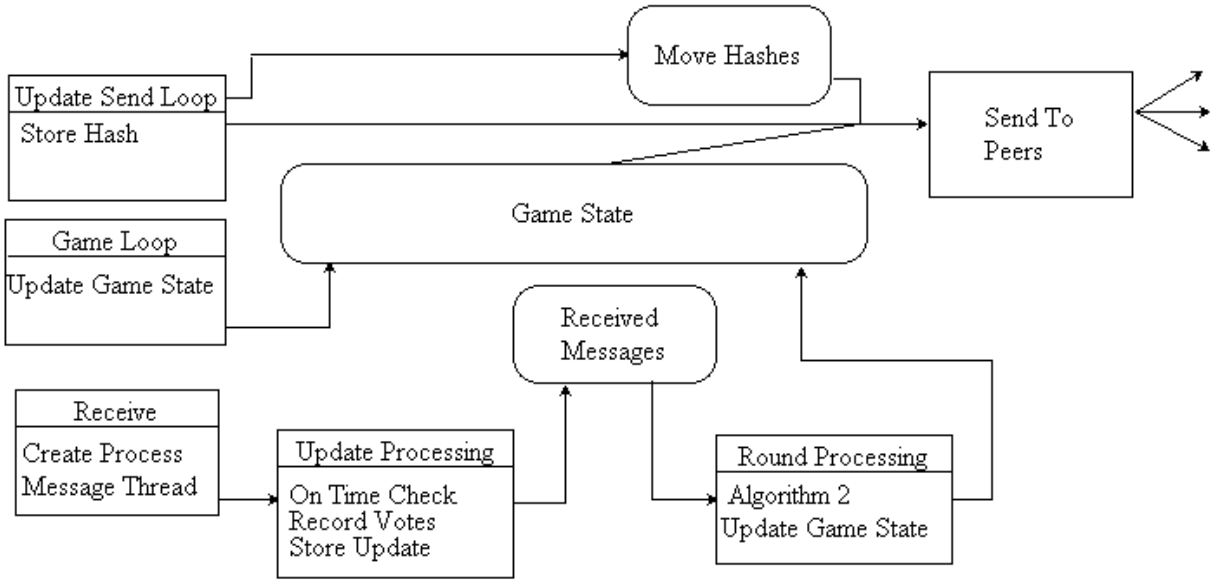


Figure 7: Flow chart of the implementation of the NEO protocol.

**Algorithm 2** NEO's processing an update algorithm

1: **if** # of votes is above threshold **then**
2: $plainTextMove \leftarrow Decrypt(encryptedMove)$
3: $hashOfMove \leftarrow Hash(plainTextMove)$
4: **if** hashOfMove == receivedHash **then**
5: Update game state
6: Store update for future hash reference
7: **end if**
8: **end if**
9: Remove old updates from storage

update has the minimum number of votes. If it does not have the minimum number of votes there is no need to examine to update any further. After the voting has been checked the next thing that needs to be done is to check that the hash of the update that was received with the last update is equal to the hash of the plain text received with this update. To do this we first have to decrypt the move information from the update into a plain text move. We then hash the plain text move and compare it with the hash received earlier, this is shown in lines $2 - 4$. Once the update passed this test it is ready to be used to update the state of the game and be stored so the hash of the next move can be referenced when it comes time to verify that move. The last thing that is always done is the old updates are removed from storage, this is so that the storage does not become too large.

The next thing that needed to be worked out was what to do with late messages, should they be processed, stored, or just thrown out. It was decided that late messages would be stored so the information could be used in the following rounds but nothing else would be done with them.

### 3.3.6 Time Synchronization

Timing is very important to the NEO protocol because rounds have to be sent and received within a certain amount of time. If the clocks are not synchronized the receiving client could think that the update was late when in fact it is on time. When a client needs to send an update every 50 ms it can be hard for the algorithm to get it exactly right. So an update may be sent out every 55 ms, after a short time this would result in an update being sent out just a few milliseconds before it needs to be arriving or this might result is an update being sent after it is supposed to arrive.

In order to fix this we developed an algorithm that checks to make sure the update being sent out has the right round number. If this algorithm notices that the round is starting to be offset, it will adjust the round number so that the updates are sent within the right time window. This can cause some updates to be skipped when the round number is being adjusted but this result is preferred over having all the updates arrive late, causing them all to be rejected.

We synchronize the clocks by integrating the Simple Network Time Protocol [24], as described in RFC 2030 [25], into the game. SNTP is a protocol that uses a time server to tell the client how far off its clock is from the server. Since SNTP is not perfect our game will use SNTP to contact the time server several times and average the responses. This makes the clocks better synchronized and allows more accurate measurements of latency.

| Variable | Lab | Distributed |
|---|---|---|
| Number of clients | $2-4$ | $2-5$ |
| Use of encryption | yes/no | yes |
| Round length | 50, 100, 250, 500, 1000 | 50, 100, 250, 500, 1000 |
| Pipeline depth | 1, 2, 3, 4, 5, 10, 20 | 1, 2, 3, 4, 5, 10, 20 |

Table 1: NEO Experiment Parameters

## 4 Experimental Setup

We conducted the experiments on a local area network in our lab environment and over the Internet. For the lab experiments we emulated the delays and losses one would typically experience over the internet. This allows us to run experiments in a controlled environment where they are easier to validate and debug.

### 4.1 Data Collection

These experiments have a number of parameters that control NEO's behavior. These parameters are shown in Table 1.

In the lab, we ran two sets of experiments, one with encryption and one without encryption,to examine the effect of encryption on update processing time. In these experiments we observe that the average time to encrypt or decrypt an update is about 2 milliseconds. Because this does not change the overall performance of the protocol, our results use only the experiments without encryption.

We vary the round length and pipeline depth of NEO so we can test a range of arrival delays. The arrival delay is the time between updates arriving at the client; it can be obtained by dividing the round length by the round depth. We only use settings such that the arrival delay is as least 50 ms. There is no need to have an arrival delay below 50 milliseconds because the client only updates the screen every 50 milliseconds. We also vary the number of players in the game between $2-5$. We did not go to 6 clients in the distributed experiments because the protocol was already having significant difficulties with 5 clients so there was no need to go on to 6.

For each experiment we save a trace of all game activity. At the end of each

| | |
|---|---|
| Update sent time | Time the update left the source client |
| Update arrival time | Time the update arrived at the destination client |
| Update size | Size in bytes of the update |
| Update source client id | The id of the client that sent the message |
| Update sequence number | The id of the update unique to the client which sent the update |
| Time message was processed | The time the processing of the message started |
| Time processing message takes | The amount of time in milliseconds taken to process the message |

Table 2: Update information that is saved in the trace of the experiments

| | |
|---|---|
| Was the update on time | Boolean value stating if the update arrived at the client within the round length. |
| Was the update accepted | Boolean value stating if the update was accepted by the client |
| Encryption delay | Time in milliseconds it takes to decrypt the update |
| Update round and depth | Used in place of a sequence number, used to determine the order of the updates |
| Update vote array | List of clients that this update is voting for |

Table 3: NEO Update information that is saved in the trace of the experiments

game, a client sends its trace information to the bootstrap server, which then saves the data to a file for inspection. These traces contain the information show in Table 2, and Table 3.

In addition to this information, the trace files contain a log of where the client is located and where the client thinks the other players are located. This allows us to compare logs and compute the location errors and to determine which settings are the best at keeping the game states equal across clients.

| | |
|---|---|
| Not on time vote | The majority of clients say the update arrived late |
| Bad hash | The update plain text did not hash to the same value as received in the previous round. |
| Do not have the move | The move never arrived |
| Do not have the move before this one | The move before this one did not arrive so there is nothing to compare it to |
| Not processed | The client did not have time to process the update |

Table 4: Reasons an update can be rejected by NEO.

### 4.2 Lab Experiments

We conduct lab experiments on two 3Ghz Intel machines with 1GB of RAM. These computers are connected through a 100Mbps local area network. Each computer runs a maximum of 3 clients and the games are played using the modified A* algorithm described earlier.

Since there is typically no loss or delay on a local area network, we emulate wide area network conditions based on data from the NLANR [26] site, which measures Internet activity between several sites throughout the United States. The emulation module emulates network delay and loss rates by using an exponential distribution around the average latency obtained from NLANR to artificially delay the updates. The average latencies range from 14 ms at the server to 45 ms. The loss is emulated using a random number generator, if the number is lower than the average number of lost packets obtained from NLANR the update is dropped.

We install a launching program on each computer and have it connect to a server. The server then sends a message to the client machine telling it to start a game, along with the parameters of the game. These parameters are shown in Table 5.

When the launching program gets a message from the server it writes out an application configuration file that contains the parameters of the game, then launches the game process.

| | |
|---|---|
| Number of clients to start | Each client machine can start more than one client, this is used in the lab experiments in the distributed experiments only one client per machine was used. |
| Game length in minutes | Number of minutes the game will last. |
| Round length | The length of each round for this game. |
| Round depth | The depth of each round for this game. |
| Is peer-to-peer | Boolean value stating if this is a peer-to-peer game or a client-server game. |
| Use dead-reckoning | Should dead-reckoning be used. |
| Use encryption | Should encryption be used, in the distributed experiments this is always true. |
| Emulate delay and loss | Should the network be emulated |

Table 5: Parameters sent to each client at the start of each game.

### 4.3 Distributed Experiments

We conduct distributed experiments so that the game could be observed in play over the Internet. The NEO protocol is designed to be resilient to the loss and delay that happens on the Internet, so the distributed experiments were conducted to show how well NEO performs. We run the distributed experiments using the same launcher program to start the experiments and the same AI to play the game.

We distributed the game by asking friends and family to install it on their home computers. People who install the game have to be able to forward the UDP ports being used for the game from their cable modem to their computer. The installer creates a registry entry on the machine so that the launcher program starts when the computer is started.

The machines in these experiments are all relatively new, and they all run on high speed connection such as cable or DSL. The computers are:

- Intel 3 Ghz - 512 Mb RAM
- Intel 3 Ghz - 1 Gb RAM
- Intel 3 Ghz - 1 Gb RAM
- Intel 2.8 Ghz - 1 Gb RAM

- Athlon 1 Ghz - 512 Mb RAM

## 5 Results

NEO is able to sustain playable game throughput when there are 3 or less clients and the arrival delay is greater than 62 ms. This is because NEO has a hard time scaling to many more clients than 3 and a higher arrival delay causes problems such as high location error which can make for poor game play. The bandwidth requirements at these settings are around 125 kbps, which can be accomplished using high speed internet connection.

However, if the number of clients is raised to 4 the bandwidth usage jumps to about 250 kbps. This is starting to be too high for most connections to maintain, which would cause the game to be unplayable because messages will get dropped and the location error will go up. Also, if the arrival delay is dropped below 62 ms the most updates will not make it to there destination on time, causing them to be rejected by the NEO algorithm.

We evaluate the results of our experiments by looking at the statistics that were gathered. These statistics show us that NEO has problems with short arrival delays causing packets to be rejected. The distributed experiments show us that highly variable internet delays cause the clients to have to hold onto updates while they wait for all updates to arrive; this causes long playout delays. We also see that lab experiments are possible, as long as the emulation accounts for latency, processing speeds, and connection speeds. We also see that NEO has a problem scaling beyond about 5 clients.

The experiment metrics that we used are location error, lost updates, unused updates, playout delay, latency, waiting delay, process delay, and arrival delay. Location error is the difference in pixels from where a client's character is and where it should be. This is caused by errors in dead-reckoning. Lost updates are the number of updates that are sent but do not arrive at their destination. Unused updates are updates that are not used due to being lost or rejected by the NEO algorithm. Playout delay, shown in Figure 8, is the amount of time from when an update leaves the source client to when it is finished being processed at the destination client. Latency is the amount of time, in milliseconds, from when an update leaves the source client

| | |
|---|---|
| Location Error | The difference in pixels from where a clients character is and where it should be. This is caused by errors in dead-reckoning. |
| Lost Updates | The number of updates that are sent but do not arrive at their destination. |
| Unused Updates | Updates that are not used due to being lost or rejected by the NEO algorithm. |
| Playout Delay | The amount of time from when an update leaves the source client to when it is finished being processed at the destination client. |
| Latency | The amount of time, in milliseconds, from when an update leaves the source client to the time it the time it arrives at the destination. |
| Waiting Delay | The amount of time between when an update arrives at its destination and when the update starts being processed. |
| Process Delay | The amount of time it takes a client to process the update. |
| Arrival Delay | The time between when two messages arrive at a client. In NEO it is found by dividing the round length by the round depth. Different arrival delays are created by varying the round length and round depth. |

Table 6: Experiment Metrics

to the time it the time it arrives at the destination, this is the first portion of the playout delay. Waiting Delay is the amount of time between when an update arrives at its destination and when the update starts being processed, this is the second portion of playout delay. Process Delay is the amount of time it takes a client to process the update and is the final portion of the playout delay. Arrival Delay is the time between when two messages arrive at a client. In NEO it is found by dividing the round length by the round depth. Different arrival delays are created by varying the round length and round depth.

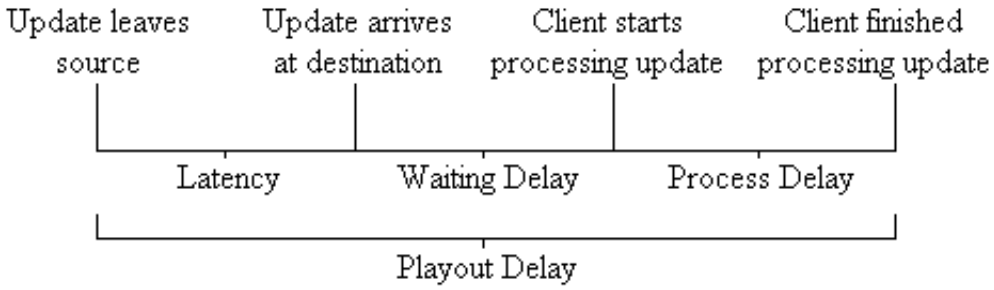


Figure 8: The time an update is in the system is broken into latency , waiting delay, and process delay; together they add up to the playout delay.

### 5.1 Lab Experiments

The lab experiments use two machines, with several clients on each, that are connected through a LAN. Since there is little to no latency in a LAN, we use emulation to simulate the internet environment. As discussed earlier, a multiplayer network game needs low latency to keep the game playable from the users perspective. In the client-server model, there is no waiting delay, so that low latency directly translates into low playout delay. This allows a client-server game to update the game state faster, so that clients all have the same game state. The goal for NEO is to keep playout delay as low as possible, while avoiding a centralized server and still keeping the game secure.

#### 5.1.1 Short Arrival Delays

One requirement of a real-time game is for updates to arrive quickly so that the user perceives real-time actions. In NEO the arrival delay is given by dividing the round length by the pipeline depth, so a game with a round length of 250 ms and a depth of 5 has an arrival delay of 50 ms. Ideally, the arrival delay should be as small as possible, however this increases the amount of unused updates and uses more bandwidth. To examine how much we can reduce arrival delay, we examined

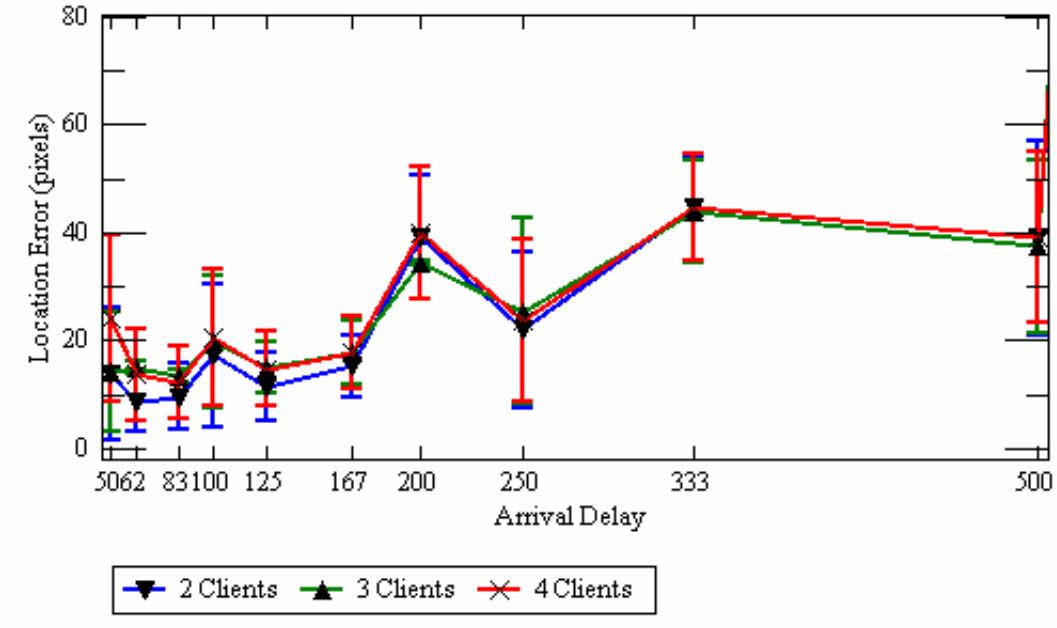


Figure 9: Location error, NEO LAN experiments

all combinations of round length and pipeline depth with a resulting arrival delay greater than or equal to 50 ms.

The main benefit of having a lower arrival delay is that the location error goes down. Figure 9 shows location error averaged over all players as a function of arrival delay. While there is some variability, in general the location error decreases as the arrival delay decreases. The variability is caused by the arrival delays coming from different combinations of round lengths and depths. For instance an arrival delay of 250 ms may result from a round length of 250 ms and a depth of 1 or from a round length of 500 ms and a depth of 2 or from a round length of 1000 ms and depth of 4. In other cases, the arrival delay comes from a unique combination of parameters. The arrival delay of 333 ms is only created by a round length of 1000 and a depth of 3. As a result arrival delays that are composed of lower round lengths have less variability. At 50 ms the location error starts to climb because there are a lot of lost and unused updates at this setting.

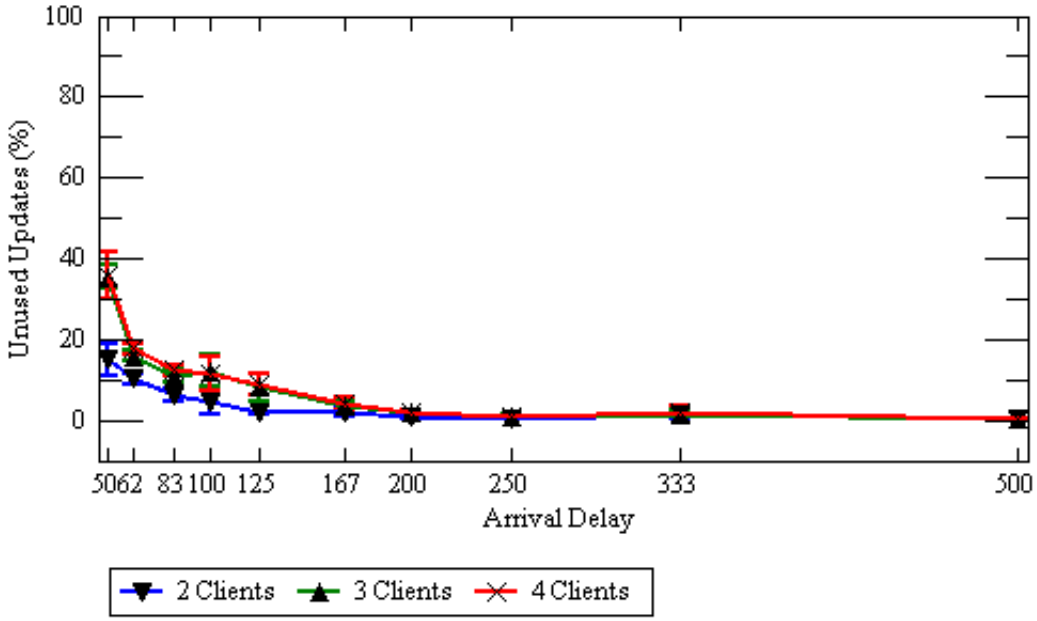


Figure 10: Percentage of unused updates, NEO LAN Experiments.

Figure 10 shows the percentage of unused updates for NEO in these experiments with 2, 3, and 4 clients. With 4 clients, the percentage of unused updates spikes up to about 35% when the arrival delay is down to 50 ms, and it is not until the arrival delay is over 100 ms that in all cases the unused updates are under 10%. This is due to the latency of the network which makes getting an update to its destination in under 50 ms difficult if not impossible; in most cases this will not be a problem as long as an arrival delay of 100 ms can be met. Figure 11 shows the average Latency for the same set of experiments. The average latency for 4 clients is about 45 ms, and the standard deviation bars show that most of the latency is between $0 - 90$ ms. Hence, the main reason that unused updates spike at low arrival delays is that the updates do not arrive on time.

Another effect of decreasing arrival delay is an increase in the bandwidth used, as more updates are transmitted per second. Figure 12 shows the rise in bandwidth for the same experiments. Notice that the bandwidth usage increases exponentially as the arrival delay gets lower, and the more clients that are used the faster the increase.

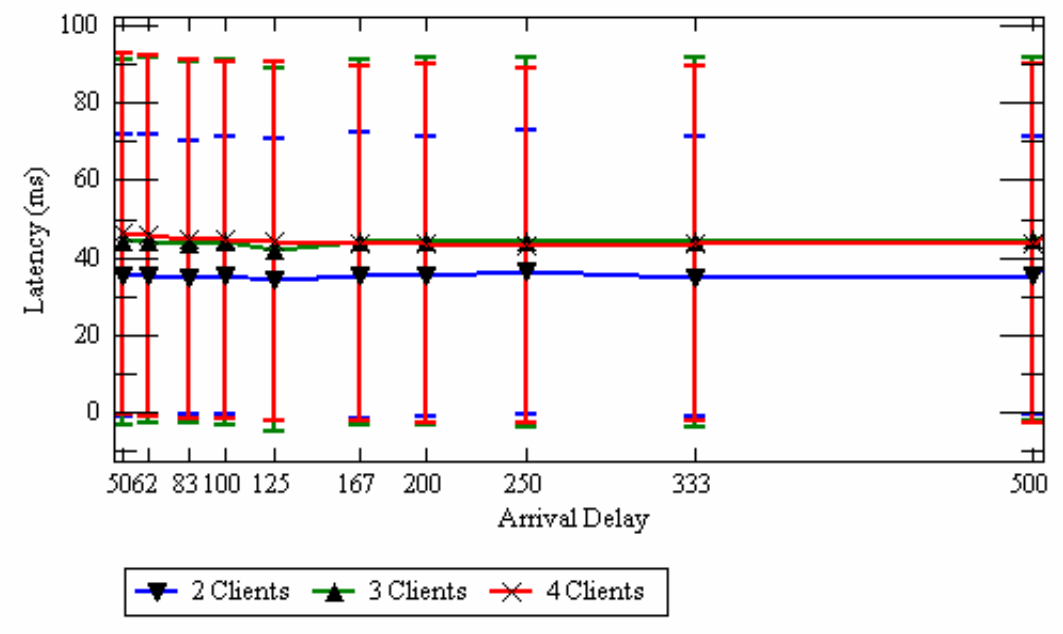


Figure 11: Average latency, NEO LAN experiments.

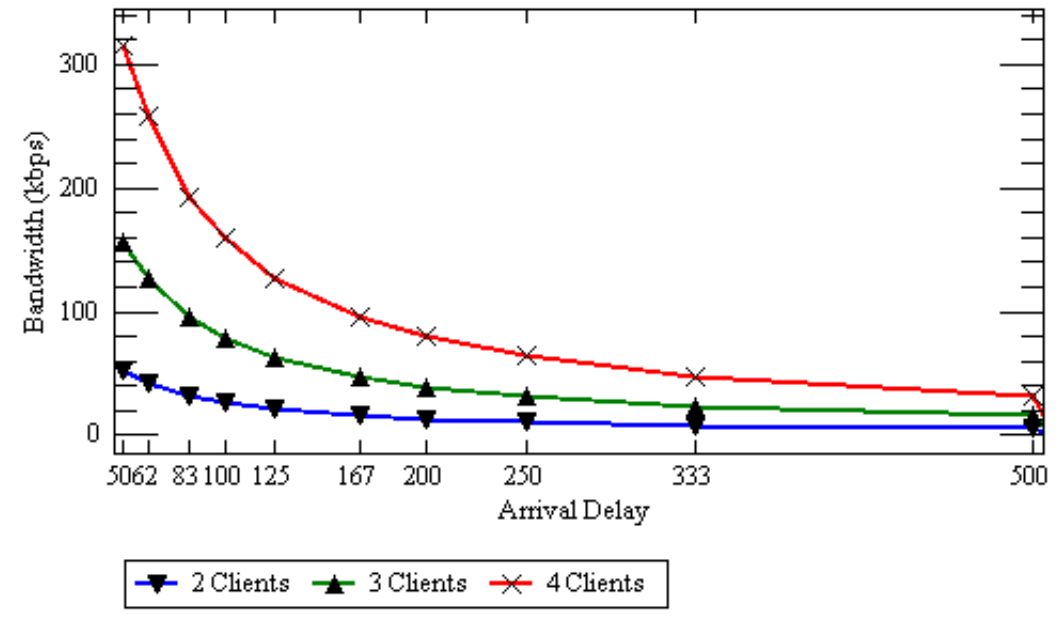


Figure 12: Total Bandwidth used, NEO LAN experiments.

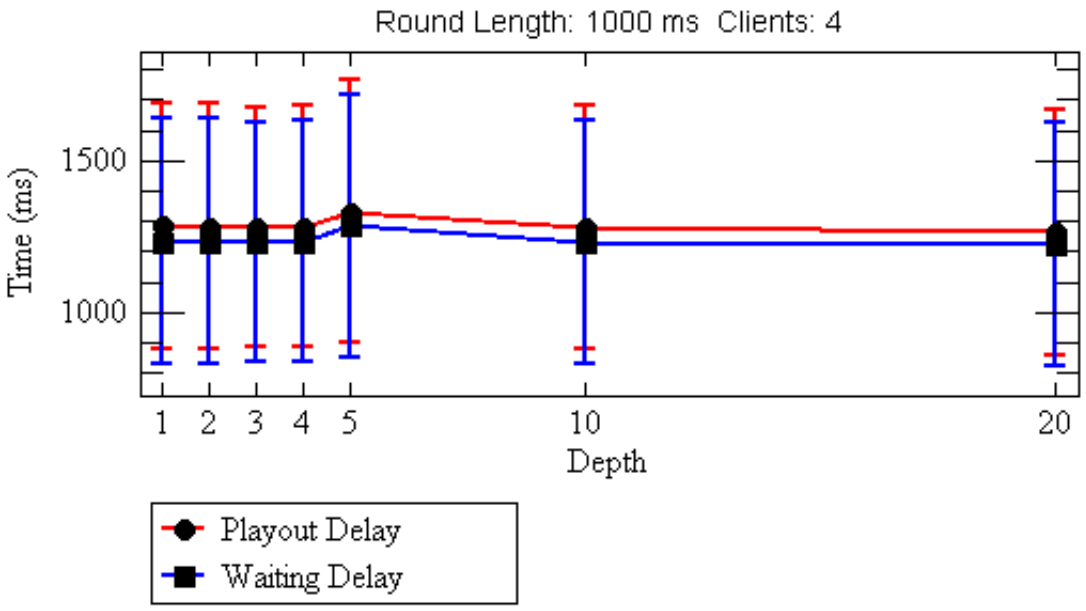


Figure 13: Playout and Waiting Delay, NEO LAN experiments.

### 5.1.2 Increasing Round Length and Pipeline Depth

One way to provide a low arrival delay, yet allow far higher latency, is to pipeline NEO updates. By increasing the round length and the pipeline depth, NEO can obtain any desired arrival delay. The cost of increasing the round length however is higher playout delays which can lead to poor game performance.

To examine the effect of increased round length, we set the round length to 1000 ms and vary the pipeline depth, using 4 clients. Making this adjustment does not affect playout delay or waiting delay because each update still has the same round length to get to the client; this is shown in Figure 13. However, as shown in Figure 14 adding depth significantly increases the percentage of unused updates, because the arrival delays are getting shorter. Offsetting this disadvantage the shorter arrival delay also causes the location error to go down. This is a huge bonus to adding depth, because a small location error makes a game more playable.

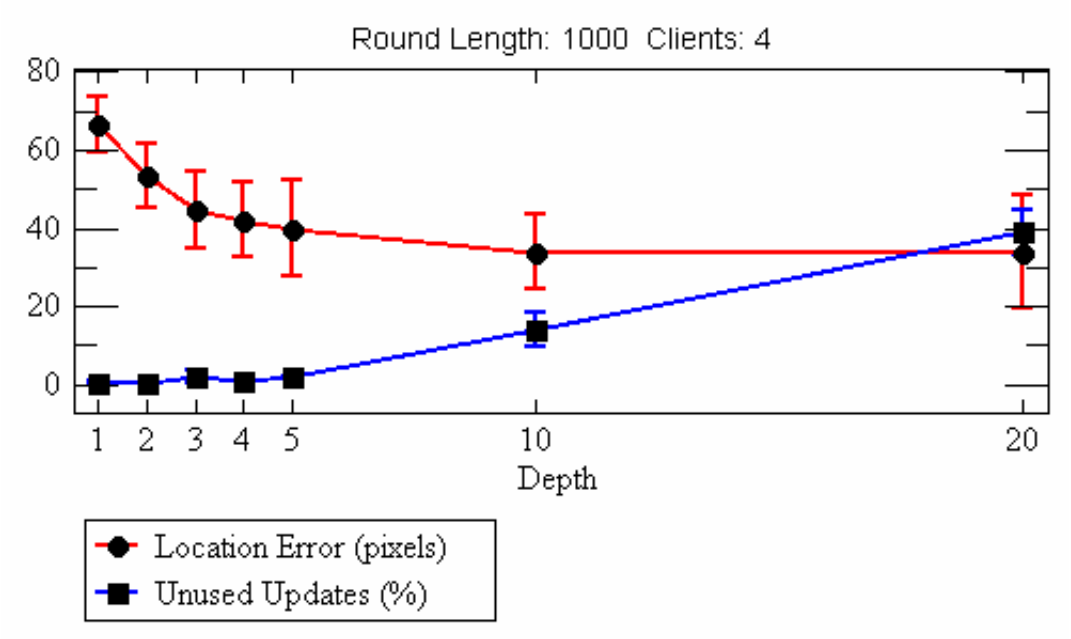


Figure 14: Location Error and Unused Updates, NEO LAN experiments.

### 5.1.3 Best Performance

The combination of settings that give NEO its best performance is hard to pin down exactly, but the preceding results suggest that we don't want the arrival delay to be lower than 100 ms and we can have a higher round length with some pipeline depth without causing bad performance in the game. With this in mind, we believe that a round length of 250 ms with a pipeline depth of 2 are the settings that give NEO the best performance in this game. This combination of settings results in an arrival delay of 125 ms, which is high enough to allow over 90% of the updates to be used, with bandwidth usage below 15 kbps. Most importantly, the arrival delay is low enough to keep the location error around 10 pixels. As shown in Figure 9 the location error is still around 10 pixels at 125 ms, but soon after it begins to climb.

The best performance for this game may not necessarily translate into the best performance for a different game. The larger an update gets, the more time it will need to get to its destination, so the round length will need to be increased. Studies

will need to be performed to determine if pipelining can continue to keep performance at an acceptable level.

### 5.1.4 Client-Server Comparison

We compare NEO to a client-server version of the game to determine if NEO is able to keep the same quality of game play as a client-server architecture. Latency is one thing that improves in a peer-to-peer architecture because the updates are passed straight to the other clients instead of through a server. The latency of the updates in NEO is not affected by changing round length or depth; while changing the round length and depth causes different arrival delays which can be seen in Figure 11, there is no change in the latency of the updates. Of course, this might change if the bandwidth gets too high, causing congestion and queueing delay in the network. This is the same as in the client-server version, except that it uses less bandwidth since each client only sends updates to the server.

NEO's location error becomes worse than client-server's location error as the number of players increases. Figure 15 shows location error for client-server and for NEO with a round length of 100 ms and a depth of 1. The location error for NEO climbs at a much faster pace than the error of the client-server, with a large standard deviation. The location error will cause the NEO version to be less enjoyable to play because the characters shown on the screen will be out of position from where they actually are.

In NEO high playout delay causes high location error. Figure 16 shows a reason why the location error is increasing for the NEO version; the playout delay is higher. Having a higher playout delay causes the client to not know where it should render the other clients character for a longer amount of time, so the client has to guess for a longer amount of time causing it to be wrong more often. This longer playout delay is caused by NEO having to wait to process the message until it has received all the messages for that round because it needs to have the votes from all the messages, while the client-server version can just process the messages as soon as it is received. This is shown in Figure 17 along with the processing delay which is also

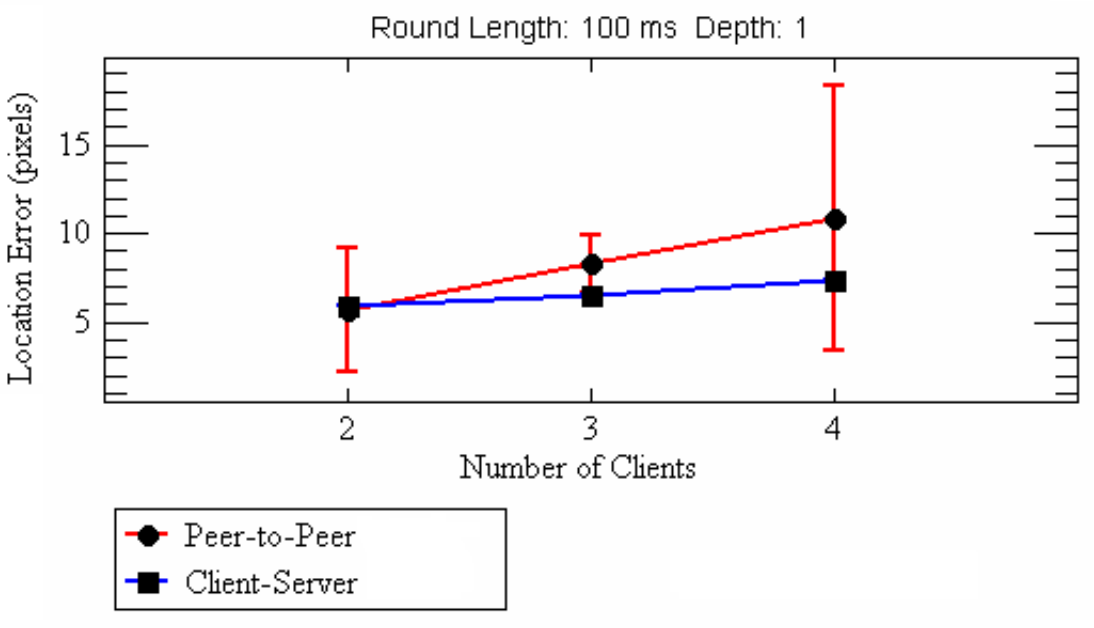


Figure 15: Average Location error, LAN experiments

always a few milliseconds longer for NEO due to the use of hashing and encryption.

To further illustrate the problem NEO has with playout delay, Table 7 shows some interesting comparisons of NEO and client-server. This table shows the time each part of the process takes with playout delay being a sum of the latency, waiting, and processing time. The table shows a round length of 250 and a depth of 2 with 2, 3, and 4 clients on both NEO and client-server. As expected, NEO has a lower latency for all clients, but NEO's waiting delay is is 30 - 40 times higher, which causes the playout delays to be nearly double. While the peer-to-peer NEO implementation saves some time in transport, it loses too much time having to wait.

### 5.1.5 Scalability

One of the primary concerns for NEO is scalability with regard to the number of players. NEO was designed to accommodate groups of $10 - 50$ players, with larger games handled through an event hierarchy [27]. However, our experiments indicate that playing the Eater game is feasible only for groups of 5 or less due to bandwidth

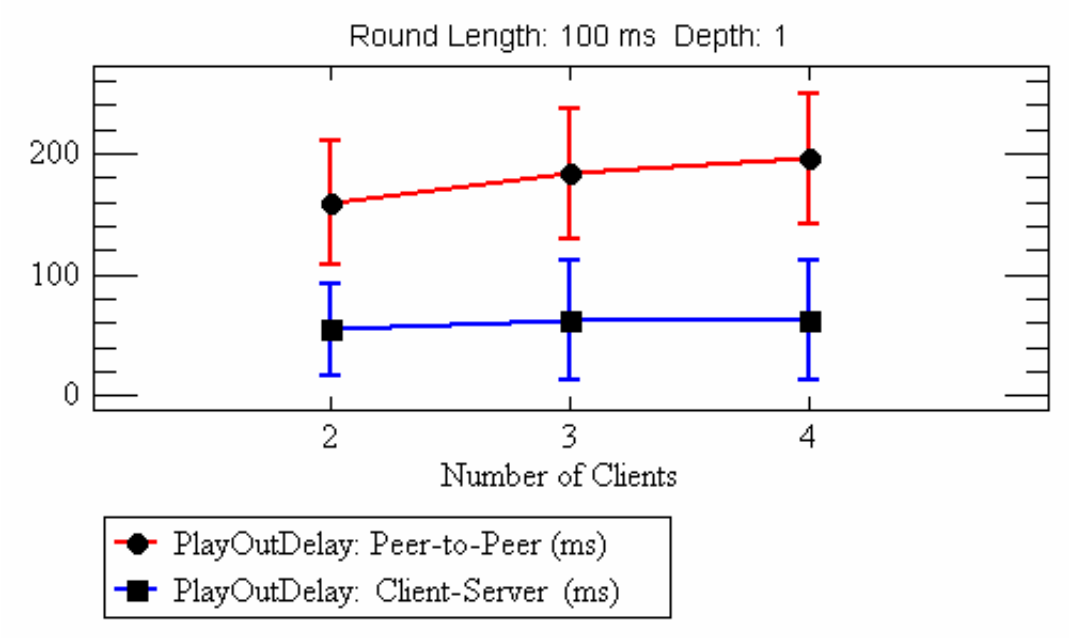


Figure 16: Playout delay, LAN experiments

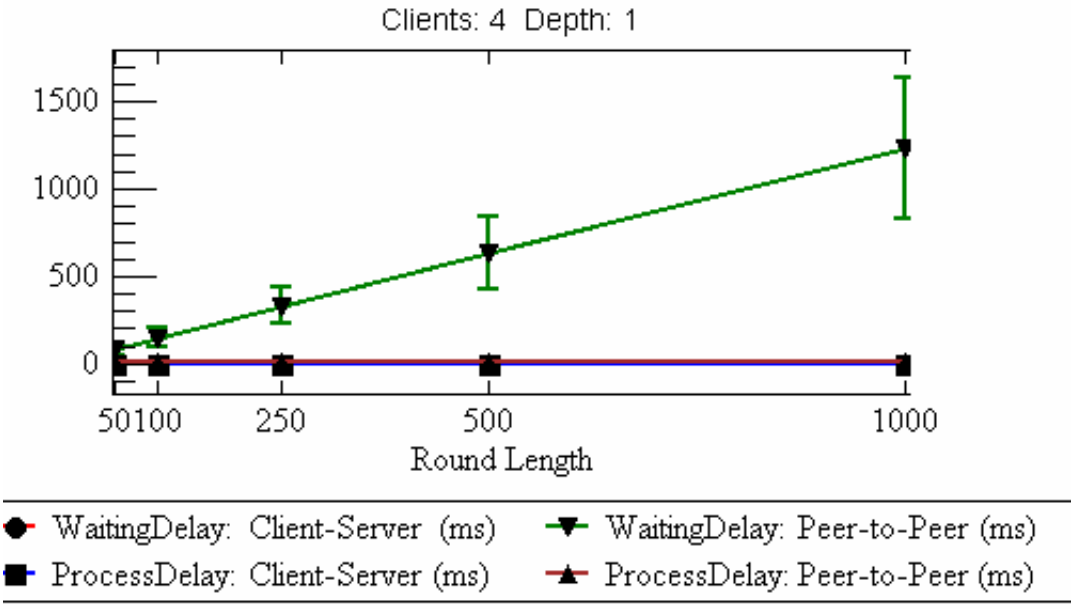


Figure 17: Waiting delay and Process delay, LAN experiments

| | Clients | Latency | Waiting | Processing | Playout |
|---|---|---|---|---|---|
| NEO | 2 | 34 | 50 | 3 | 87 |
| | 3 | 42 | 65 | 4 | 111 |
| | 4 | 44 | 80 | 4.5 | 128.5 |
| Client-Server | 2 | 48 | 2.2 | 0 | 50.2 |
| | 3 | 59 | 2.4 | 0 | 61.4 |
| | 4 | 61 | 2.3 | 0 | 63.3 |

Table 7: This is a break out of where the average time is being spent for a round length of 250 and a depth of 2.

| Clients | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|
| Kbps | 13.5 | 20.67 | 31.25 | 44 | 66 | 99 | 148.5 | 222.75 |

Table 8: Bandwidth usage, upload and download, for different clients with arrival delay of 125 ms

usage and high playout delay.

Taking a closer look at NEO with an arrival delay of 125 ms we see that NEO does not scale very well. In Figure 12 it can be seen that NEO is behaving as would be expected of a basic peer-to-peer protocol with the bandwidth nearly doubling for each client added. Using these numbers Table 8 extrapolates what the bandwidth usage would be for even more clients. It is interesting to note that with just 4 clients the game is already having trouble being played over a modem connection, which typically runs at $< 56$ Kbps. Although, on a DSL or cable connection the game will be able to scale to about 9 clients before it starts having difficulties.

Since the Eater Game is a relatively small game with small updates, we look at an update for a hypothetical larger game. Table 9 shows the size of a NEO update used in our experiments and what could be the size of a NEO update in a larger game. When a game between 2 clients is run for 5 minutes approximately $2,400$ updates are sent with an arrival delay of 125 this is a rate of 8 per second. Using the larger update from Table 9 this results in 21.9 kbps. This would obviously be a very big problem if it was scaled to many more clients.

In addition to bandwidth, processing delay and ultimately playout delay also increase slightly as more clients play the game. Recall that process delay is the amount

| Item | Eater Game | Larger Game |
|---|---|---|
| ID | 1 | 2 |
| Round # | 4 | 4 |
| Depth # | 4 | 4 |
| # Votes | 1 | 1 |
| Votes | 2 | 10 |
| Hash length | 1 | 1 |
| Hash | 20 | 20 |
| Move length | 1 | 1 |
| Move encrypted | 128 | 300 |
| Total(bytes) | 162 | 343 |

Table 9: Size in bytes of a NEO update message.

of time it takes the client to get the information it needs out of the update, from the time it starts looking at an update to the time it is done looking at the update. Figure 18 shows that while the processing delay does not change with different round lengths, it is always higher when more clients are added. This is because there are more votes to be recorded and the client machine is doing more work at the same time causing the processing to take longer overall.

Increased processing delay also contributes to a higher playout delay. Recall that the playout delay is the amount of time it takes from when the update is sent to when it is done being processed. As shown in Figure 19, the playout delay increases by about 20 to 30 ms when the number of clients goes from 2 to 4, this is due to several factors including longer waiting and process delays. If this trend continues, as more clients are added it will be very difficult or impossible to get messages sent and processed under 500 ms, which would make for a very bad user experience.

The higher processing and playout delays cause a ripple effect through the game. The next area that feels this ripple is that of unused updates. An unused update is an update that was either lost in transit or arrived but was not used due to several reasons: not enough time to process the update, the update may have shown up late, not enough votes, or the update was rejected by NEO. Figure 20 shows the percentage of unused updates for NEO with a round length of 100 ms and a pipeline depth of 2. Updates are sent every 50 ms, which causes many more unused updates

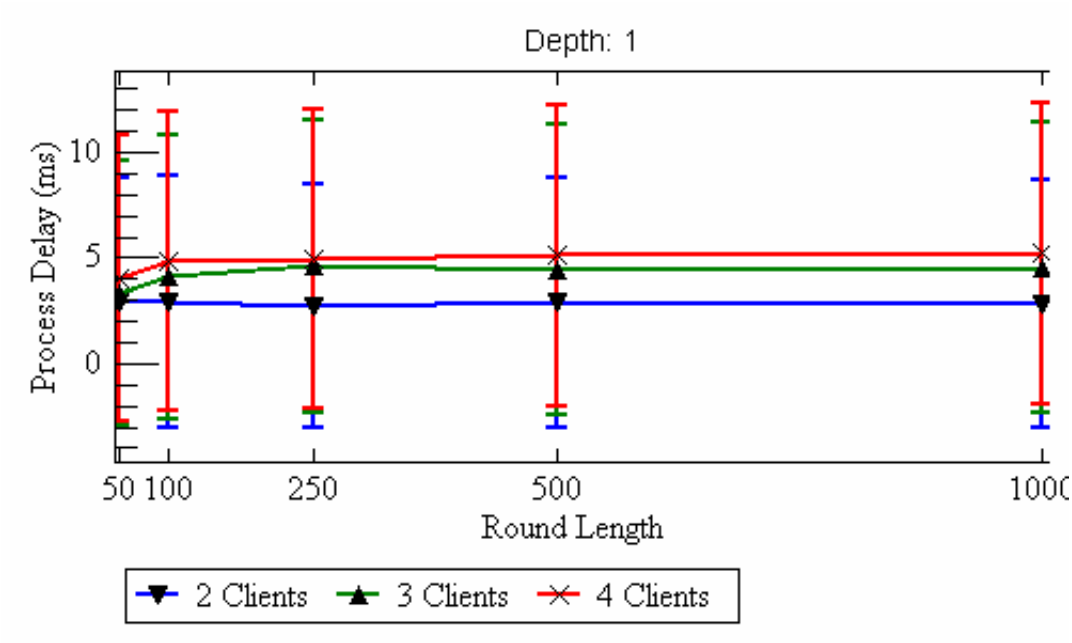


Figure 18: Process Delay, NEO LAN experiments.

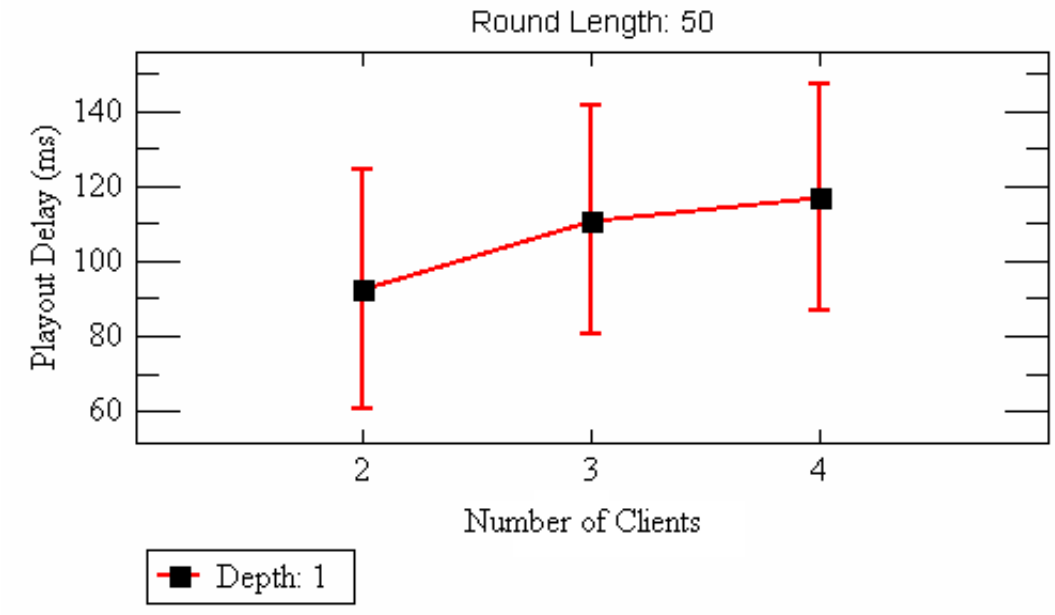


Figure 19: Playout Delay, NEO LAN experiments

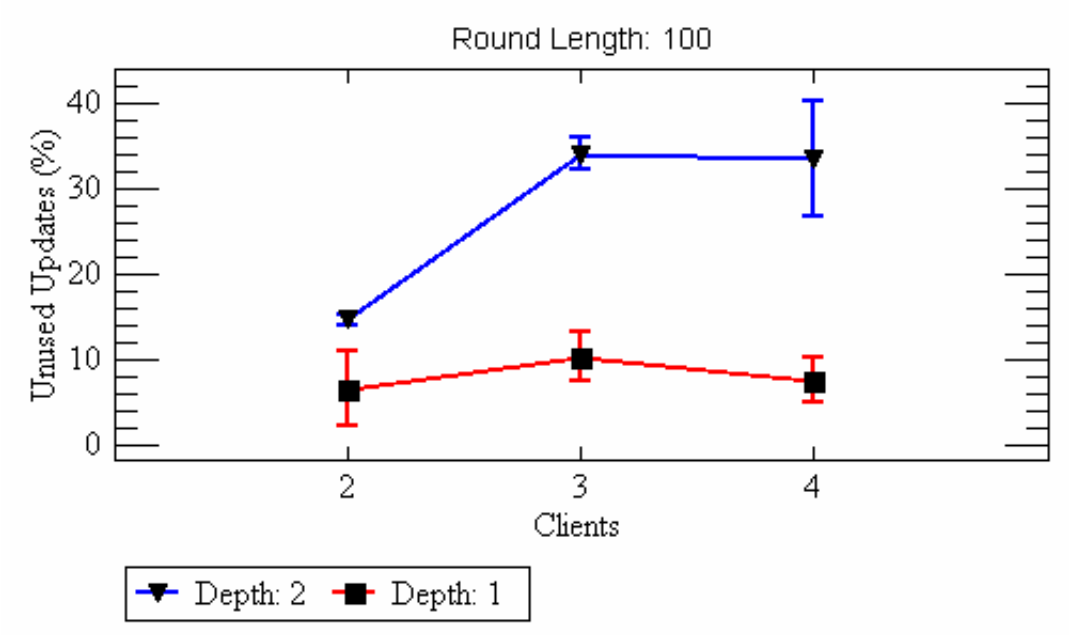


Figure 20: Unused updates, NEO LAN experiments

than sending updates every 100 ms. This is because the extra 50 ms allow more time for the latency and processing of the update and so they all have a chance to be used. Using 100 ms arrival delay causes under 10% of the updates to be unused which is reasonable for the game we were using, but when a 50 ms arrival delay is used the unused updates jump to over 30% which would be unreasonable for most any game. This amount of unused updates causes the game to become unplayable because the state of each client is too much different from each other. The amount of unused updates also causes strain on the network that does not need to be there because the updates being sent are not being used.

All of these scalability problems finally result in a location error issue. Location error is the amount of pixels a dead reckoned client is off from where it really is. Shown in Figure 21, with 4 clients the location error jumps to almost 30 pixels. The extent of this location error problem depends on the game that is being played. For the Eater game used in these experiments an error of around 30 pixels can be annoying but still playable. For a first-player shooter game an error that high could result in targets

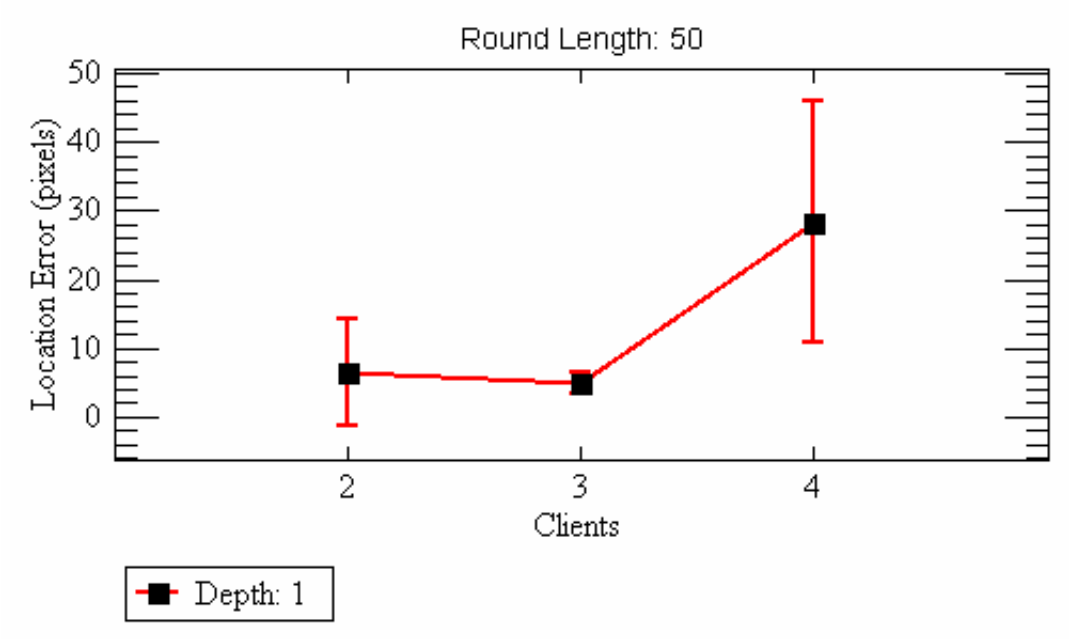


Figure 21: Location Error, NEO LAN experiments

being missed when they looked like they would be hit, making the game unplayable.

## 5.2 Distributed Experiments

The distributed experiments were run on several machines within the state of Utah, each using a different Internet Service Provider. The experiments were conducted so that the results could be compared to the results obtained in the lab and so that NEO could be further tested.

### 5.2.1 Short Arrival Delays

In the distributed experiments NEO is able to run with lower arrival delays while keeping unused updates low. Shown in Figure 22 the unused updates only has a problem when the arrival delay dropped to 50 ms. This indicates that the average latency between the clients was lower than 63 ms but higher than 50 ms. Figure 23 shows latency by arrival delay, while the standard deviation is very high the average does stay below 60 ms except for the 5 clients set, but this one stays below 70 ms. This

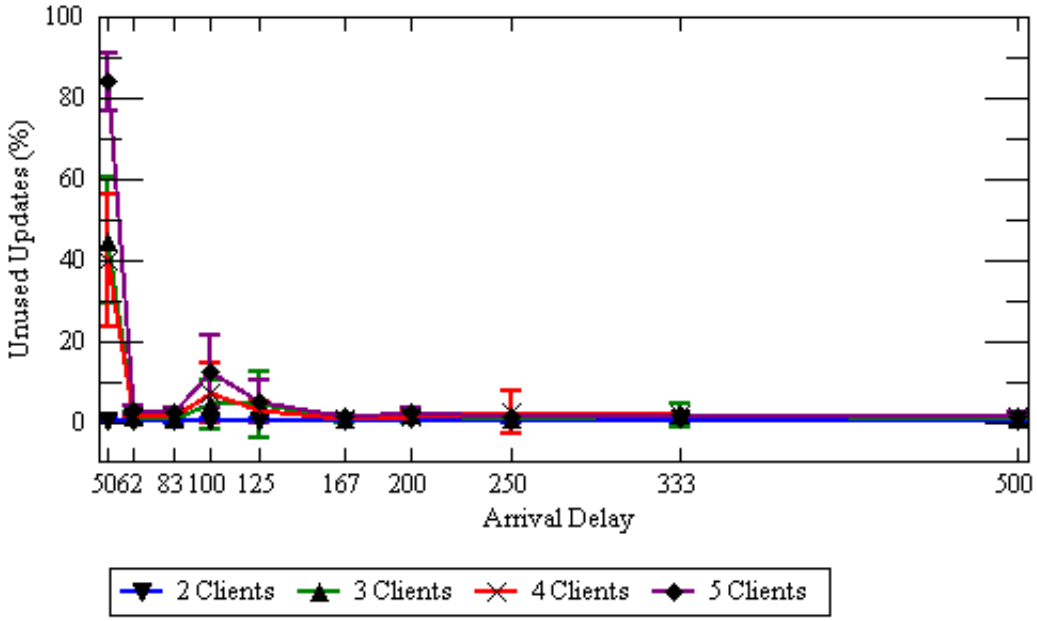


Figure 22: Unused updates, NEO Internet experiments

slightly higher latency is probably more a factor of the clients that were being used than it was a problem of scalability, because the higher latency happens at all arrival delays. If the higher latency was due to the increased number of clients we would expect the latency to go down as the number of clients went down. This result shows us that the emulation module used in the lab experiments was emulating latency correctly but it may have been dropping too many updates causing more unused updates.

While the percentage of unused updates was better for the distributed experiments the location error was higher. Shown in Figure 24, at an arrival delay of 125 ms the location error for the distributed experiments was around 30 pixels, but on the LAN experiments the error is only about 15 pixels. This is due to the waiting delay being higher during the distributed experiments. The average waiting delay is not much higher in the distributed experiments than it was in the lab experiments although it does have a higher standard deviation. The higher standard deviation causes a higher waiting delay because if just one update is coming in slowly the entire

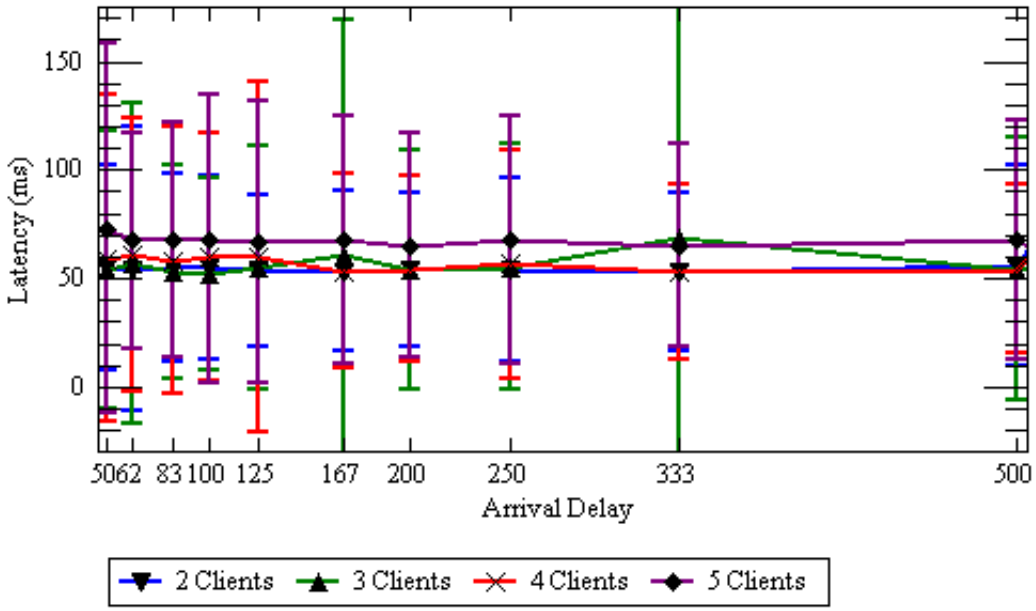


Figure 23: Average latency, NEO Internet experiments

group has to wait for the slow one.

### 5.2.2 Increasing Round Length and Pipeline Depth

Again, we use round length and pipeline depth to try and get a low arrival delay while allowing for higher latencies. Figure 25 shows how increasing the round length affects the playout delay we see that with 5 clients as the round length went up the playout delay was always as high as two round lengths with very low standard deviation. We then looked at the effect of depth on playout delay with a round length of 1000 ms. Shown in Figure 26, changing the depth has little on affect on the playout delay and the playout delay of 5 clients is just above 2000 ms which is the length of two rounds, so changing the depth has no affect on the playout delay.

Having to wait for 5 clients causes the playout delay to be the same as the round length which in turn causes high location error. There are three components of playout delay; latency, waiting delay, or processing delay, any of which could cause a high playout delay. As shown in Figure 27 the latency and process delays stay constant

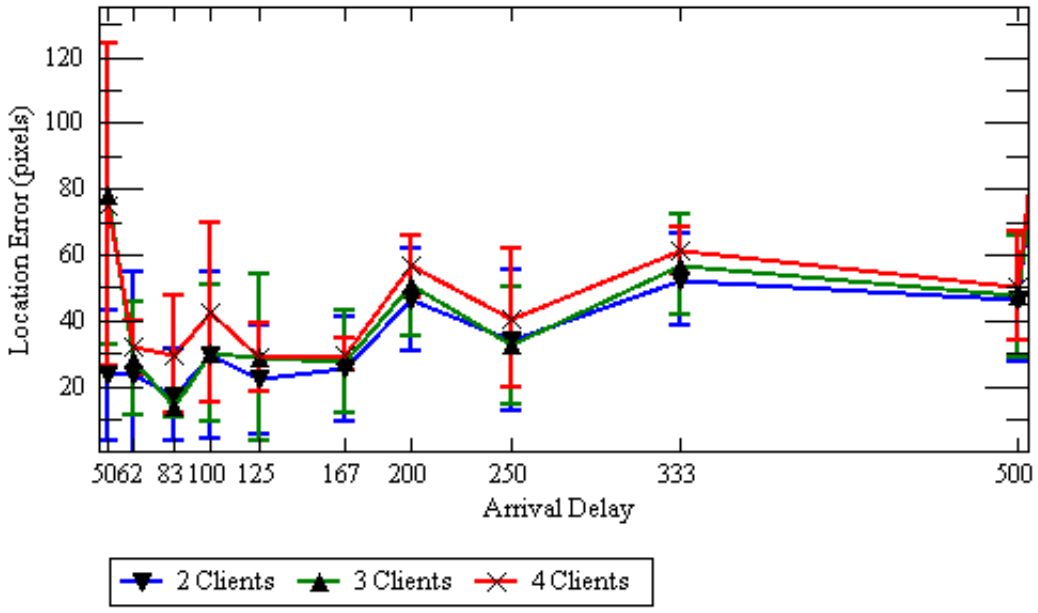


Figure 24: Location Error, NEO Internet experiments

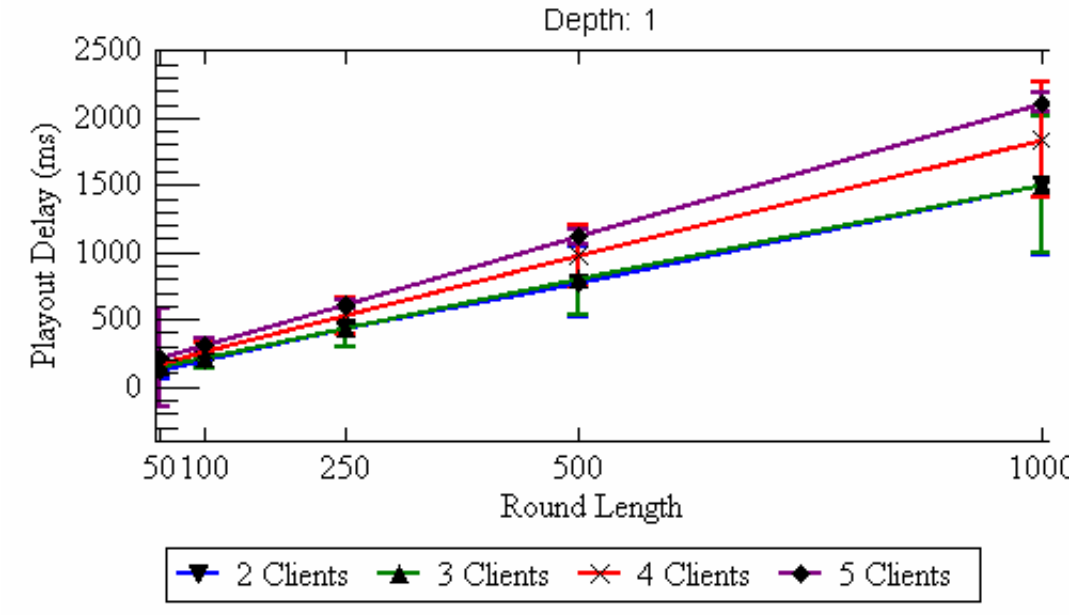


Figure 25: Average Playout delay by round length, NEO Internet experiments

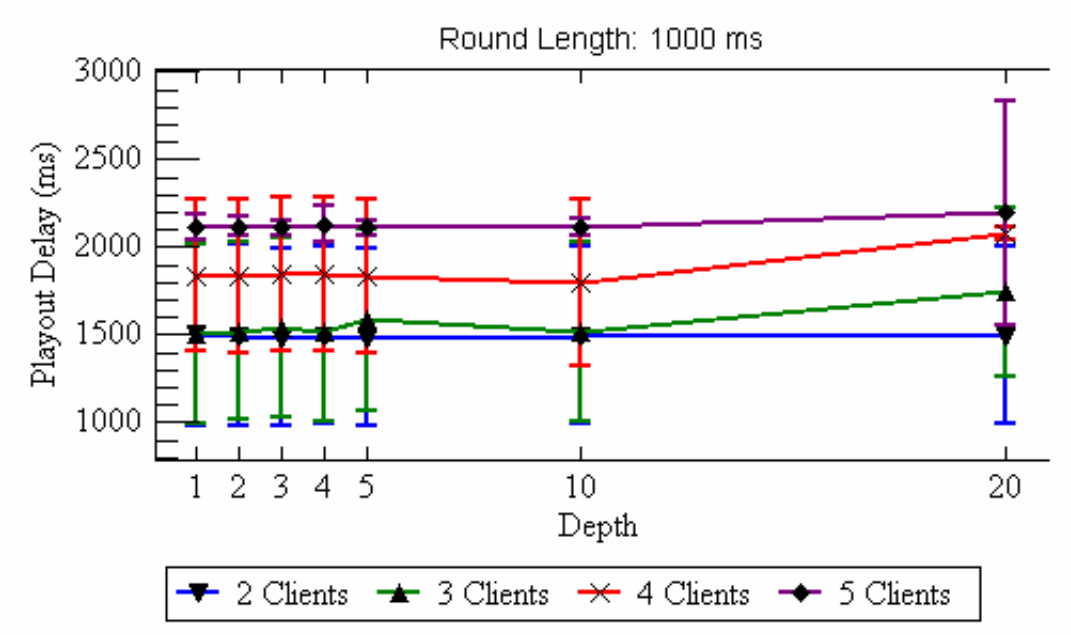


Figure 26: Average Playout delay by depth, NEO Internet experiments

while the waiting delay increases with the round length. Again, this happens because as more clients are added there is a greater chance that one of the updates will take the full round length to arrive or that one update will not arrive. When this happens the clients must wait until the round is over before beginning to process the other updates.

For a round length of 1000 ms there is high variance in playout delay until the arrival delay is dropped to 50 ms. Shown in Figure 28, the average points generally have no change for the first 6 depths but they have a high standard deviation, so at times they could be performing quite well. When the depth reaches 20, taking the arrival delay to 50 ms, the points jump up about 200 ms and the standard deviation drops to almost nothing, meaning that it is usually going to take between 1900 and 2000 ms. This is again because the arrival delay was too low for the computers to be able to process the updates faster than they were arriving.

High playout delay can cause high location error and unused updates. Next we looked at the percentage of unused updates and location error associated with

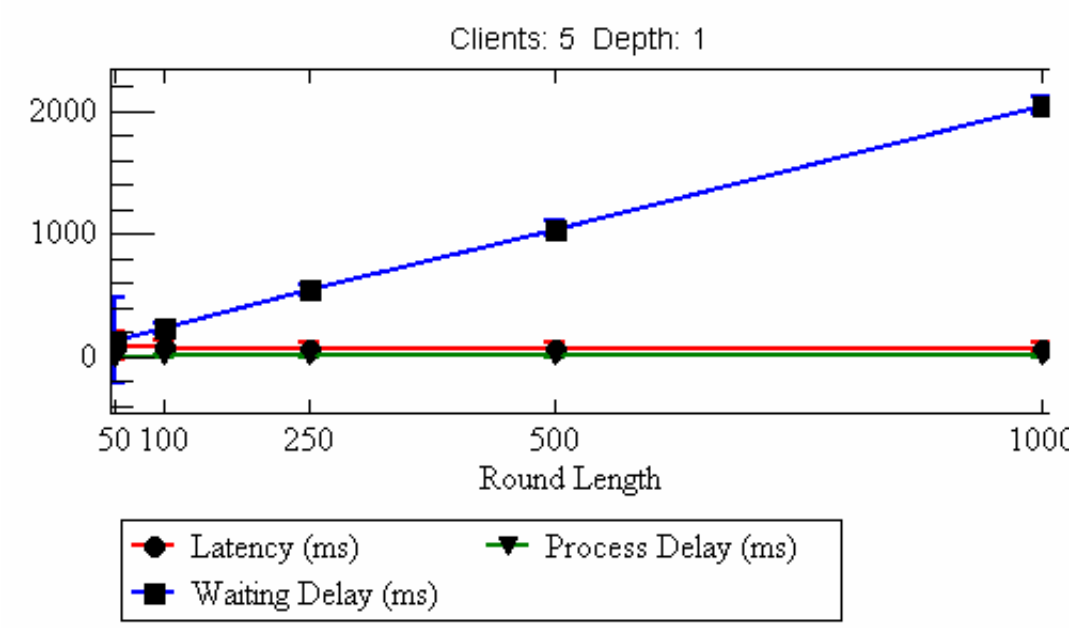


Figure 27: Multiple stats by round length with a depth of 1 using 5 clients.

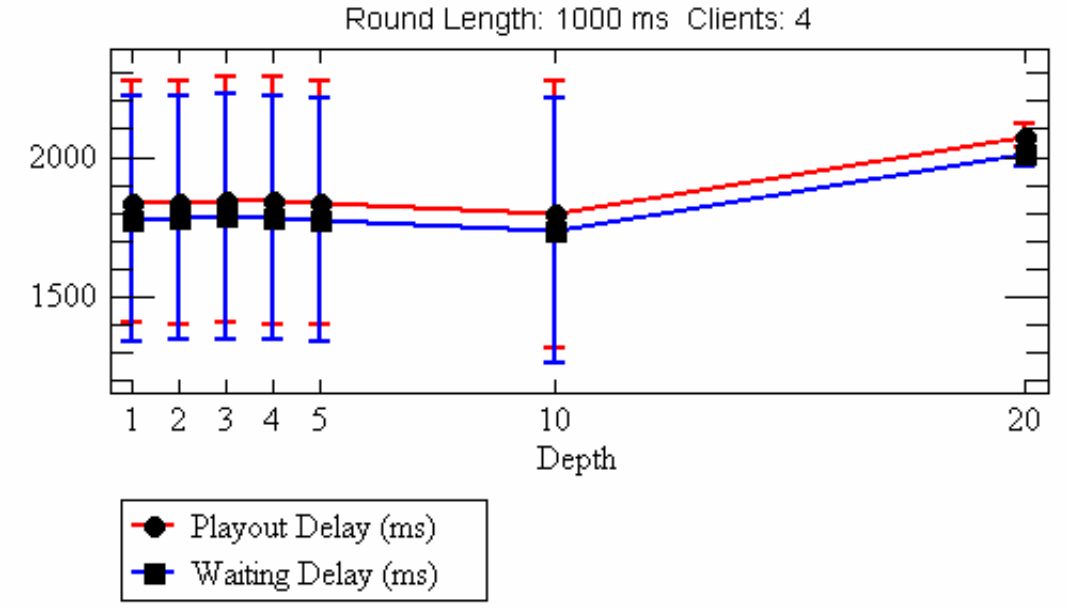


Figure 28: Waiting and playout delays by depth with a round length of 1000 ms and 4 clients.

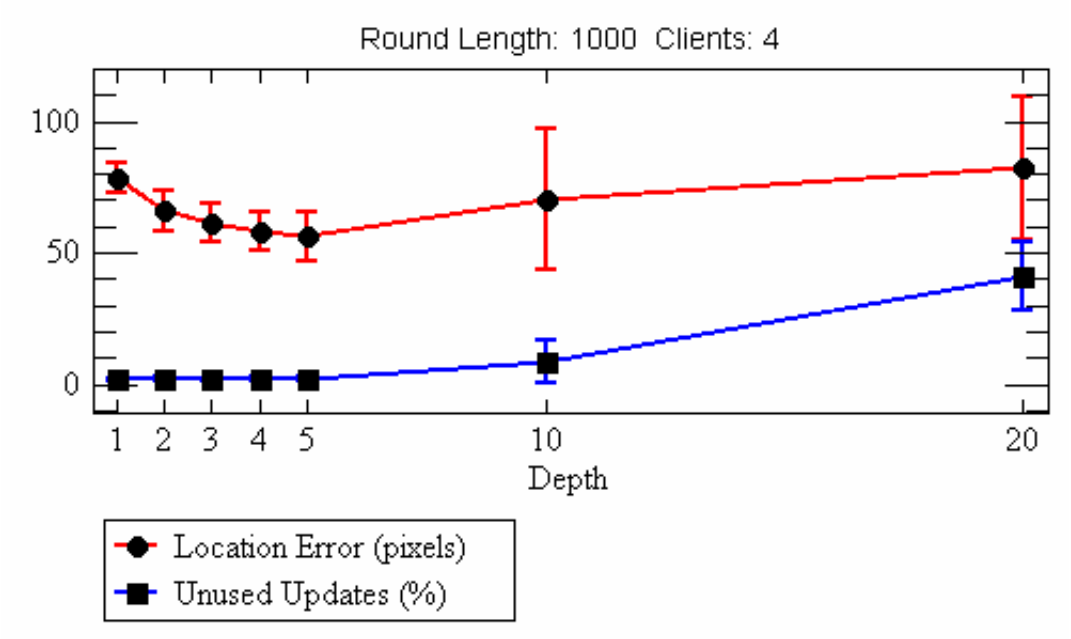


Figure 29: Location error and unused updates by depth with a round length of 1000 ms.

increasing the depth of a 1000 ms round. Shown in Figure 29 the results were different for the distributed experiments than they were for the lab experiments. We can see that increasing depth causes the location error to go down until a depth of 10 which is when the number of unused updates begins to climb. This result is expected because as the number of unused updates goes up the client will have to use dead reckoning more often which can raise the likelihood of there being a location error. Making the depth 20 with a 1000 ms round length actually causes more location error than just having a depth of 1, which means the client only sends an update every second.

### 5.2.3 Best Performance

It is again very hard to pin down one group of settings that is the best, for these tests no single group performed the best in all cases. We want a group of settings that keeps the round length and the arrival delay as small as possible without going so low that it starts to affect performance. In the case of the distributed experiments

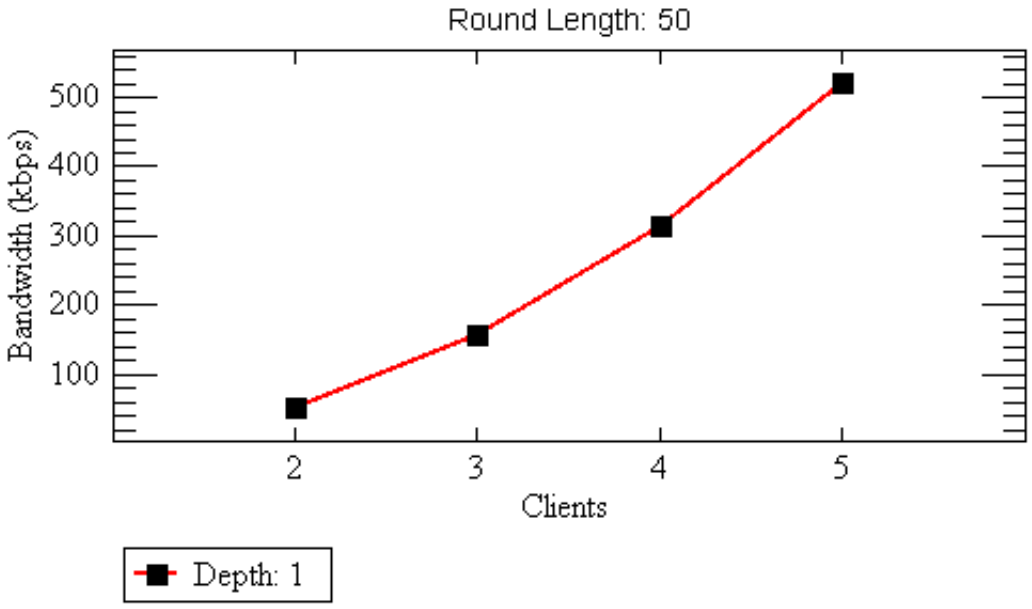


Figure 30: Bandwidth, NEO Internet experiments

we have shown that the point were round length and arrival delay really start to affect performance is when they are below 62 ms so any setting with a round length close to that would work well.

### 5.2.4 Scalability

The bandwidth usage in the distributed experiments once again shows that NEO has serious scalability problems. Figure 30, shows a worst case for NEO, an arrival delay of 50 ms, we see that with 5 clients NEO is approaching the bandwidth limit of most home high speed connections. As discussed earlier the amount of bandwidth used is lower at an arrival delay of 125 ms, but NEO still has trouble scaling beyond 10 clients.

As we have talked about earlier adding more clients causes the waiting delay to go up. Figure 31 shows the waiting delay by round length with a pipeline depth of 1, it is shown that having just 5 clients causes the average waiting delay to be around the length of the round with very low standard deviation. This is a problem

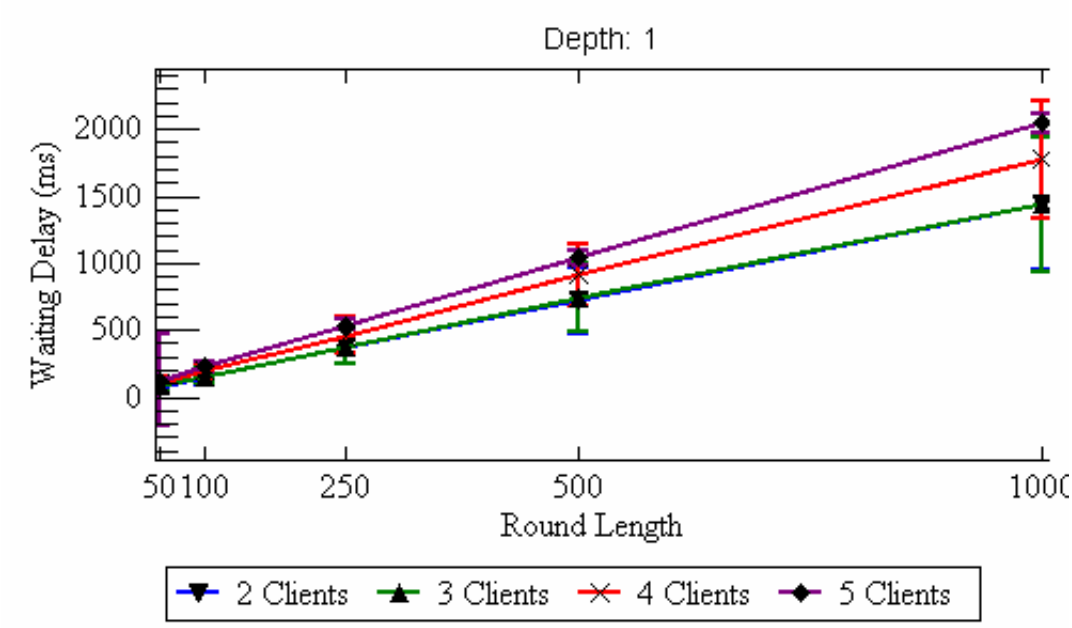


Figure 31: Waiting delay by round length, NEO Internet Experiments

because we have seen that raising the round length can help NEO to perform better as long as the depth is also increased. However, Figure 32 shows the waiting delay by depth for a round length of 1000 ms, the waiting delay does not get better by adding depth and in some cases it actually makes the delay worse. This shows that the two main factors that cause waiting delay are the number of clients and the length of the round.

Longer waiting delays cause a higher playout delay which causes other problems like location error. Shown in Figure 33, as more clients are added the playout delay goes up and the standard deviation gets smaller. This smaller standard deviation means that the playout delay is more likely to be closer to the average and since the average is higher it will cause more problems like high location error.

A high number of unused updates can cause the game to become unplayable due to differences in the game states between clients. Shown in Figure 34, a low arrival delay, the depth of 2, and an increasing number of clients contribute to a high percentage of updates going unused. Also, with an arrival delay of 100 ms, increasing

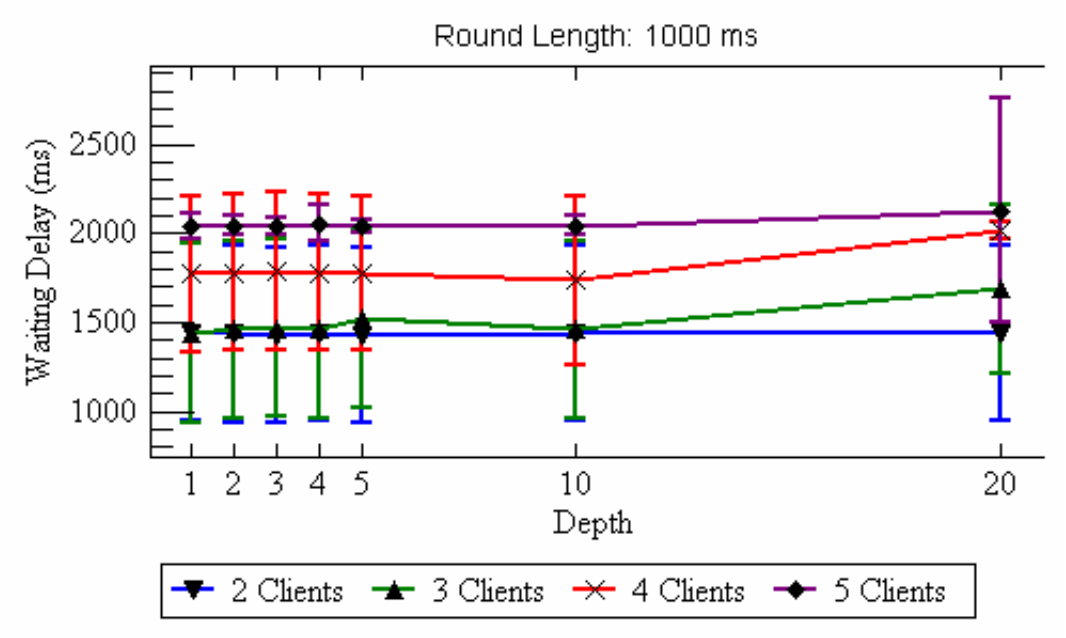


Figure 32: Waiting delay by depth, NEO Internet Experiments

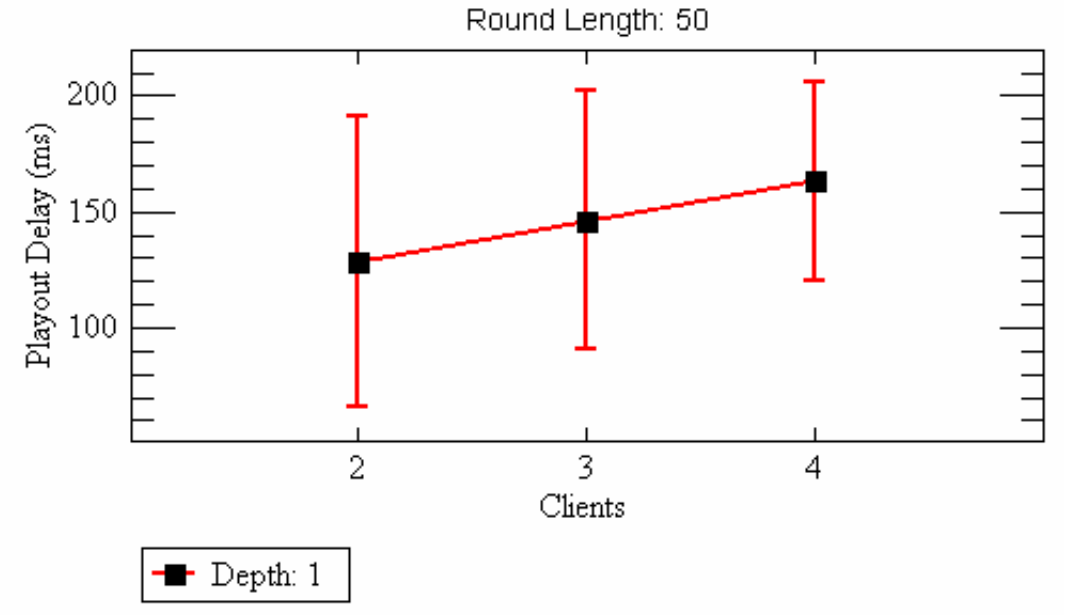


Figure 33: Playout delay, NEO Internet experiments

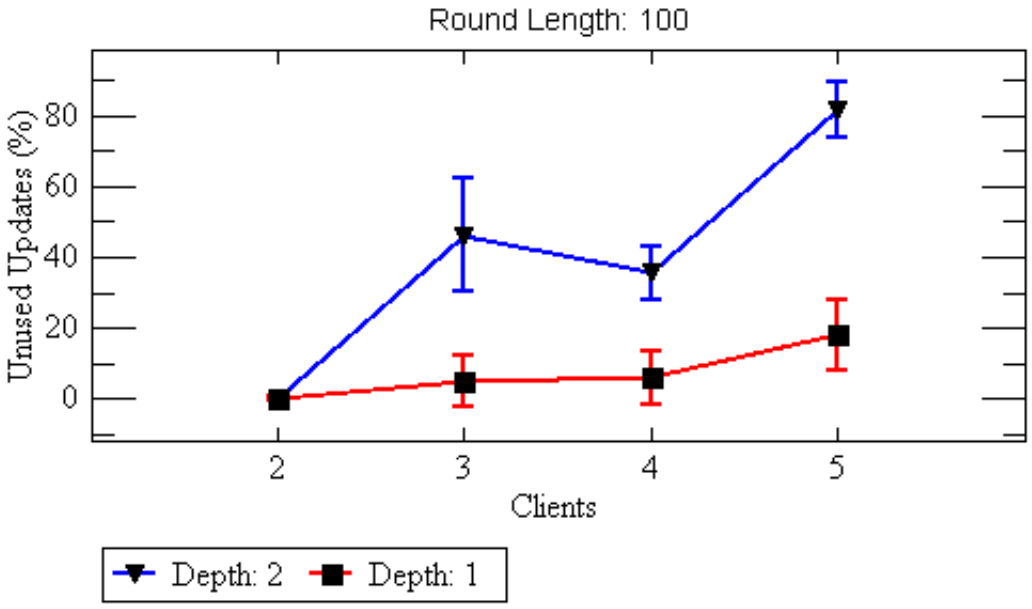


Figure 34: Unused updates, NEO Internet experiments

the number of clients still causes the percentage of unused updates to rise, especially from 4 - 5 clients. At 5 clients the unused updates reaches 20% which is pushing at the limit, anything higher than that will cause a serious problem for the game because it will not have enough updates or consistent updates.

These scaling problems are finally manifest in the form of location error. Shown in Figure 35, the location error of 4 clients is around 100 pixels while the location error of 5 clients is about 200 pixels. As discussed earlier this can be corrected by lengthening the round length, but longer round lengths also cause problems with location error, so there is no easy solution. Although as we have shown it is better to stay away from a very low arrival delay especially if there is going to be more than 3 or 4 clients.

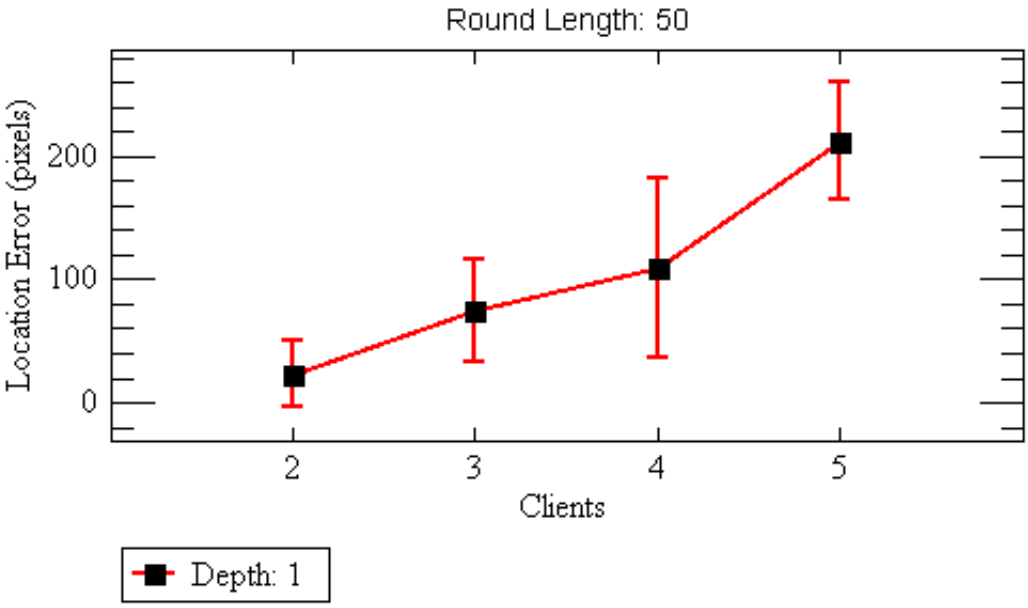


Figure 35: Location Error, NEO Internet experiments

## 6 Conclusions

We have found that NEO is able to run effectively when $2-4$ clients are used and the arrival delay is short enough to keep the location error down, while being long enough to allow the updates to get to the client in the given time. Our results show that an arrival delay of 62 ms to 125 ms performed the best but the best performance will be different for each game.

Our results also show that NEO has a problem scaling beyond 4 clients. After 4 clients the bandwidth usage starts getting so high that most high speed Internet connections would not be able to support the game. Using a higher number of clients also causes NEO to not use updates because they are late, or the client does not have enough time to process the update. These unused updates lead to location errors that are so high they make the game unplayable.

Lowering the waiting delay is the best way to improve the playout delay of the NEO protocol. As mentioned earlier a system of processing an update as soon as it has enough votes could lower the waiting delay but it might not lower it enough to make a significate difference. In this case the next best thing would be to keep the round length down, because the waiting delay is never much higher than the round length.

Both sets of results, lab and distributed, were similar enough to show that a simulator can produce results very close to those that exist in the real world. The simulator will need to be able to simulate different connection speeds and latencies as well as simulating different computer speeds. We saw in these experiments that computer speeds do matter because one of our distributed test machine was considerably slower than the rest of the machines and it had the most problems keeping up when the arrival delays were low. Also the simulator should be run on more than one machine which communicate over a LAN because it is hard for one machine to simulate enough clients to determine if the protocol is having trouble scaling.

Overall NEO performed very well with very few updates being rejected and when updates were rejected it was because the round length was so short that the updates could not get to their destination on time, or the arrival delay was so low

that the client was being overwhelmed with updates.

The item that caused the most problems for NEO was the waiting delay. Updates spent a lot of time sitting at the client not being processed especially with a long round length and a larger amount of clients. Something needs to be done to lower this waiting delay, this can be done by keeping the round length low or allowing an update to be processed when it has enough votes to be accepted instead of waiting for all the other updates to arrive.

Finally, NEO has a significant amount of overhead such as larger packet size, waiting for rounds to be done to process moves, and problems getting UDP port forwarding setup. These items seem to take away from the bonuses that a peer-to-peer real-time system gives. In the end this causes NEO to be right on par with client-server architectures because they both have pluses and minuses. Each solution has to be looked at carefully to see if it is meeting the requirements of the project well enough for the benefits to outweigh the negatives.

### 6.1 Protocol Issues

From our experience building a game using NEO, we have made a number of observations. First, the main benefit to using UDP over TCP is that if an packet is slowed down or lost for some reason the next packet UDP is delivered as soon as it arrives, whereas when TCP is used the next packet will not be delivered until the missing packet has been requested and arrives. This is a large benefit to a game that needs fast updates more than it needs constant updates. Such is the case in the game used in this work, if it misses one update it can continue on without it and the game play is barely affected, unless it is an important move like one that changes the state of a stone.

The primary drawback of using UDP is that it forces the application developer to design reliability and congestion control for the application. Using TCP solves this problem because it already has many of the mechanisms that are being developed for NEO, such as message recovery, congestion control, and working through a firewall or a router. In addition most personal computers today do not have public IP addresses,

instead a modem or router has a public IP and it assigns a private IP to the computer. This is not a problem for a protocol like TCP which establishes an outgoing connection through the firewall which can then be used to receive incoming messages through the firewall but for a connectionless protocol this causes the packets to have to be forwarded from the router to the computer. In order to forward a packet from the router to the computer the user needs to enter the information into the router, which can be a daunting task for someone who does not feel comfortable around computers. Currently many games use TCP as their network transport so that users do not have to change settings in their router. This could be the biggest drawback of the NEO protocol because users do not want to have to change settings on their computer every time they want to install a new application.

Another problem with NEO as talked about earlier is that many updates spend a lot of time just sitting at the destination waiting for the rest of the updates to arrive. When a 4 or more clients are used the playout delay is always the same length as the round length. This causes higher playout delays and in turn causes high location errors which lead to poor game play. This is not a problem with client-server because updates are processed as soon as they arrive at the client.

Another challenge for NEO is clock synchronization. Clock synchronization is very important to NEO and can be much harder to accomplish than it seems. If the games clocks are not synchronized NEO will believe that some updates are late when they are not or NEO will believe that updates are on time when they are not. In our game we used the SNTP to find out how far off the client machines clock was from the real time. This gives an accuracy of within 5 ms which is good enough to keep NEO working correctly.

The final issue that came up with NEO is a hole in the security. It was noticed that while a client can not send out false moves to other clients when using NEO a client could send out different moves to each client. This could cause the cheating player to have an advantage over the other players because it would allow him to be in more than one place at a time. While this would be a fairly complicated cheat, people playing games have been known to go to great lengths to get the upper hand

in a game. This cheat could be overcome by having client A verify the hash that was received from client B with the hash that client C received from client B.

## 6.2 Potential Improvements

With the large number of combination of NEO settings, network conditions, and game types it is hard to know what settings should be used when. We believe that a machine learning algorithm could be trained to recognize the network and game conditions and adjust the round length and depth to an optimal setting. Some research would need to be done to determine what are the optimal settings for different kinds of network and game conditions. When this work is completed it will make implementing NEO into a game much more effective.

One issue that needs to be worked on is making NEO consume less bandwidth as it scales to more clients. The use of a multicast network is a possible solution to this problem. Multicast would give NEO greater scalability but it would have a problem with security, because the updates would be passed through intermediate nodes. More research would be needed to come up with a solution for this.

Earlier we talked about different peer-to-peer systems such as Mercury which try to make peer-to-peer games scalable by using a DHT. It seems that a combination of NEO and a DHT could actually perform quite well, NEO having the security and the DHT having the scalability. If NEO was to use a DHT for its node management and message passing infrastructure scalability would not be an issue and NEO could still have all the security it needs.

There is an extension to NEO being worked on that would allow for greater scalability. This extension includes creating NEO groups, these groups would grow in size from the clients in the same room with you to all the clients in the game. This would make NEO more scalable because a client would not have to send every outgoing update to every other client in the game. Instead each client would only send the updates to the clients that are part of the group that the update affects. Although our research shows that the group which receives the most updates will not be able to be much bigger than 5 clients.

The last item that can be looked at is to compare and contrast a TCP version of NEO with the UDP version. As mentioned earlier TCP has many features built into it that NEO is implementing. It would be interesting to see if TCP can accomplish that goals of NEO while keeping playout delay and location error down. This would make NEO easier to implement and install and therefore more likely to be used.

# 7 Appendices

## 7.1 The Mono Project

The Mono Project [28] is an open source implementation of the .NET architecture. It was used to compile and run the application on Linux machines. The reason for wanting to use the Mono project was to have this application put onto Planet Lab [29]. Planet lab is a group of Linux machines distributed around the world which would be a perfect test bed for this application. Unfortunately, the Mono project is still a very young project and so it has some problems that made it impossible to use and still get reliable results.

The first problem that was run into with the Mono Project was that it will not compile on a network share. This was easily fixed by moving the project to a local directory, but the real issue was that the Mono Project was having trouble writing a file to a network share. While this was not an overwhelming problem it did make debugging difficult because the game writes an error log and if the game is not able to write the error to the file debugging is almost impossible.

There was another problem with the networking aspect of the Mono Project. The problem was that when trying to use the same UDP socket to send out connectionless messages to all other clients in the game, the Mono Project implementation would close the socket after it sent one message. This problem was overcome by creating a UDP client for each other client and opening a connection to that client.

The final issue is the one that lead to the implementation of an encryption flag as one of the game parameters. This is because the RSA decryption in the Mono Project took on average 300 milliseconds, compared to Microsoft .NET where the RSA decryption takes $< 10$ milliseconds. This adversely affects the playout delay of NEO and may make some of the results look worse than they really would be.

Overall, the problems that were encountered with the Mono Project made its use infeasible due to the unreliability of the results and the difficulty in debugging. So the Linux version was thrown out and the project was installed on friends machines instead.

## References


[1] E. S. Association, "Essential facts about the computer and video game industry," *http://www.theesa.com/files/EFBrochure.pdf*, 2005.

[2] A. Wierzbicki and T. Kucharski, "P2p scrabble. can p2p games commence?" in *Peer-to-Peer Computing, 2004. Proceedings Fourth International Conference on*, August 2004, pp. 100 – 107.

[3] J. Smed, T. Kaukoranta, and H. Hakonen, "Aspects of networking in multiplayer computer games," in *Proceedings of International Conference on Application and Development of Computer Games in the 21st Century*, L. W. Sing, W. H. Man, and W. Wai, Eds., Hong Kong SAR, China, Nov. 2001, pp. 74–81.

[4] J. D. Pellegrino and C. Dovrolis, "Bandwidth requirement and state consistency in three multiplayer game architectures," in *NETGAMES '03: Proceedings of the 2nd workshop on Network and system support for games*. New York, NY, USA: ACM Press, 2003, pp. 52–59.

[5] B. Ng, A. Si, R. W. Lau, and F. W. Li, "A multi-server architecture for distributed virtual walkthrough," in *VRST '02: Proceedings of the ACM symposium on Virtual reality software and technology*. New York, NY, USA: ACM Press, 2002, pp. 163–170.

[6] W. Cai, P. Xavier, S. J. Turner, and B.-S. Lee, "A scalable architecture for supporting interactive games on the internet," in *PADS '02: Proceedings of the sixteenth workshop on Parallel and distributed simulation*. Washington, DC, USA: IEEE Computer Society, 2002, pp. 60–67.

[7] S. O. E. Inc., "Everquest experiences a record number of simultaneous players," *http://sonyonline.com/corp/press_releases/eq_100k_users.html*, July 2002.

[8] C. GauthierDickey, D. Zappala, V. Lo, and J. Marr, "Low latency and cheat-proof event ordering for peer-to-peer games," in *NOSSDAV '04: Proceedings of*

*the 14th international workshop on Network and operating systems support for digital audio and video.* ACM Press, 2004, pp. 134–139.

[9] L. Pantel and L. C. Wolf, "On the suitability of dead reckoning schemes for games," in *NETGAMES '02: Proceedings of the 1st workshop on Network and system support for games.* New York, NY, USA: ACM Press, 2002, pp. 79–84.

[10] C. Diot and L. Gautier, "A distributed architecture for multiplayer interactive applications on the internet," in *IEEE Networks magazine*, vol. 13, no. 4, July/August 1999. [Online]. Available: citeseer.ist.psu.edu/diot99distributed.html

[11] N. E. Baughman and B. N. Levine, "Cheat-proof playout for centralized and distributed online games," in *INFOCOM*, 2001, pp. 104–113. [Online]. Available: citeseer.ist.psu.edu/baughman01cheatproof.html

[12] S. L. H. Lee, E. Kozlowski and S. Jamin, "Synchronization and cheat-proofing protocol for real-time mulitplayer games," in *Proc. of Intl Workshop on Entertainment Computing*, May 2002.

[13] E. Cronin, B. Filstrup, and S. Jamin, "Cheat-proofing dead reckoned multiplayer games," in *Intl. Conf. on Application and Development of Computer Games*, January 2003.

[14] C. GauthierDickey, D. Zappala, and V. Lo, "Event ordering and congestion control for distributed multiplayer games (draft)," February 2005.

[15] A. R. Bharambe, S. Rao, and S. Seshan, "Mercury: a scalable publish-subscribe system for internet games," in *NETGAMES '02: Proceedings of the 1st workshop on Network and system support for games.* New York, NY, USA: ACM Press, 2002, pp. 3–9.

[16] A. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems," *Lecture*

*Notes in Computer Science*, vol. 2218, 2001. [Online]. Available: citeseer.ist.psu.edu/rowstron01pastry.html

[17] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan, "Chord: A scalable Peer-To-Peer lookup service for internet applications," in *Proceedings of the 2001 ACM SIGCOMM Conference*, 2001, pp. 149–160. [Online]. Available: citeseer.ist.psu.edu/stoica01chord.html

[18] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker, "A scalable content-addressable network," in *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*. New York, NY, USA: ACM Press, 2001, pp. 161–172.

[19] B. Knutsson, H. Lu, W. Xu, and B. Hopkins, "Peer-to-peer support for massively multiplayer games," in *INFOCOM 2004. Twenty-third Annual Joint Conference of the IEEE Computer and Communications Societies*, March 2004, pp. 96 – 107.

[20] M. Castro, "An evaluation of scalable application-level multicast built using peer-to-peer overlay networks," 2003. [Online]. Available: citeseer.ist.psu.edu/castro03evaluation.html

[21] Y.-J. Lin, K. Guo, and S. Paul, "Sync-ms: synchronized messaging service for real-time multi-player distributed games," in *Network Protocols, 2002. Proceedings 10th IEEE International Conference on*, Nov 2002, pp. 155 – 164.

[22] D. Bauer, S. Rooney, and P. Scotton, "Network infrastructure for massively distributed games," in *NETGAMES '02: Proceedings of the 1st workshop on Network and system support for games*. New York, NY, USA: ACM Press, 2002, pp. 36–43.

[23] E. Game, *http://www.c-sharpcorner.com/Code/2002/Oct/EaterGameII.asp*.

[24] V. Bocan, *SNTP Client in C#, Implementation of the Simple Network Time Protocol (RFC 2030) in C# http://www.codeproject.com/csharp/ntpclient.asp#xx847034xx*.

[25] D. M. U. of Delaware, *Simple Network Time Protocol (SNTP) Version 4 for IPv4, IPv6 and OSI http://www.ietf.org/rfc/rfc2030.txt*.

[26] NLANR, *Measurement and Network Analysis http://watt.nlanr.net/active/amp-utah/HPC/body.html*.

[27] C. GauthierDickey, V. Lo, and D. Zappala, "Using n-trees for scalable event ordering in peer-to-peer games," in *NOSSDAV '05: Proceedings of the international workshop on Network and operating systems support for digital audio and video*. New York, NY, USA: ACM Press, 2005, pp. 87–92.

[28] *The Mono Project*. [Online]. Available: http://www.mono-project.com/Main_Page/

[29] *PlanetLab Consortium*. [Online]. Available: http://www.planet-lab.org/