Implementing a Distributed Architecture for MMO based Asteroids

DANIEL TRUONG
GEORGE ZHANG
RODRICK YU
SERGUEI MICHTCHENKO

ABSTRACT

We present a distributed architecture for a Massively Multiplayer Online version of Asteroids. Our project consists of a single cohesive world that is divided into regions, where each region is handled by a different server. In this paper we discuss key logical issues such as multiple servers hosting segments of a single world, maintaining consistency between servers, and seamless transition of objects and clients between servers.

1 INTRODUCTION

Our project attempts to address the problem of load distribution between multiple servers in a multiplayer environment. In typical client-server architecture, the server keeps an authoritative state of the environment and sends updates to players in order to simulate a single coherent environment. We have attempted to create a distributed system of dedicated servers each responsible for simulating a part of the environment whereby players are able to transition seamlessly between the segments.

In an MMO, players change the state of their character or environment by sending in the state of their player controlled object (e.g. position, rotation, and velocity) to the server which must confirm the new state and inform other players of these events. In addition to managing the state of players, the server must perform numerous computations for all of the objects in the world as they interact with each other and also update the state of non-user controlled objects. This potentially results in a CPU bound server process or an unacceptably high network latency between the server and the client. Our project addresses these issues with a distributed system of dedicated servers: more machines/servers mean more CPU available to calculate CPU bound processes and servers only sending the client data local to that server.

Asteroids is a game where a player controls a ship placed in an asteroid field. The objective is to shoot the asteroids and survive as long as possible without colliding into an asteroid [1]. Our implementation follows this basic design but adds a multiplayer component. Several players must survive as long as possible by shooting the asteroids. Note that in our implementation there is no direct player versus player (i.e. players cannot shoot each other) for the sake of simplicity.

Important technologies we use in our project include Microsoft's XNA 3.0 [2] for our client and the Lidgren Networking Library [3] for client server communication. Microsoft's XNA tools allow us to streamline the development of the client while focusing our efforts on the logical issues of creating a

distributed server architecture for MMO Asteroids. The use of the Lidgren Networking Library [3] provided us a simple C# API with several useful features in implementing client server communication.

2 RELATED WORK

There are several solutions to creating distributed architectures that exploit spatial locality of players with various ongoing research efforts in developing scalable architectures. The following are some notable papers that have influenced our decisions in implementing our distributed MMO architecture.

"Architecture for MMORPG" [4] the authors present a scalable centralized server architecture through the modularization of components. These components separate the application duties and responsibilities and thus in theory could be hosted on one or more machines. To achieve high numbers of concurrently connected players the architecture is set up so that as much computation as possible is passed off to the client machine.

"Load-balancing for Peer-to-Peer Networked Virtual Environment" [5] the authors present an alternative solution to the client-server architecture. Similar to what we have done, the authors take advantage of a player's area of interest and divide the world into various subspaces. However instead of each area being handled by a server as we have done, each area is assigned to a peer to keep the state of consistency of that area. They achieve interactions between sub-servers based on player's position. One of the problems with their architecture is that it is not cheat proof. Since the client maintains the state of a subspace, a malicious user could intentionally change that state to their benefit.

"A Distributed Architecture for MMORPG" [6] the authors present an approach to distributed architectures for MMOs that we have tried to follow closely in our implementation of MMO Asteroids. Similarly, the authors split the virtual world into smaller regions with each region handled by a different server for a single game world. They present important algorithms and techniques to reduce bandwidth requirements for the game servers and clients, address consistency, hotspot, congestion and server failure problems, and seamless interaction between players residing on areas handled by different servers.

3 IMPLEMENTED DESIGN

Our current implementation of the MMO Asteroids game consists of two servers where each server is a multithreaded process. We initially began with one multithreaded server and expanded the system to function with two physical servers. Unfortunately, due to time constraints, the two server architecture is currently unstable and not suitable for testing. Below we detail our design for the multithreaded server and the two server architecture.

3.1 MULTITHREADED SERVER

The server process consists of several threads. The server-side game engine has its own thread whose task is to call an update function at a regular interval in order to update object positions and do collision detection. This update tick is performed atomically which means any messages arriving during the update will be admitted into the next tick. There are two message pump threads: one for client-server messages and one for the server-server messages. These message pump threads feed events into the game engine, hence they block while the game engine is updating. However, blocking does not congest the network interface because the message pump thread is reading messages from a queue which is filled by another thread inside the Lidgren network library [3]. The message pump threads are not CPU intensive and the message processing bursts mesh between the ticks of the game engine when the state can be updated.

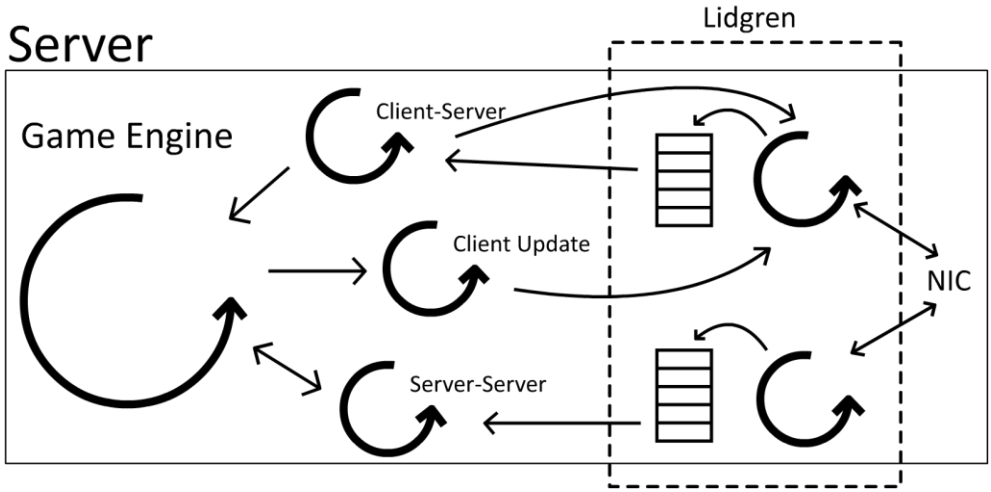

Figure 1: Multithreaded server architecture.

Another thread sends updates to each client at a fixed time interval. This thread is fairly CPU intensive because it calculates the area of interest for each client connected to the server. Here is also a potential bandwidth bottleneck because the worst case bandwidth needed is $O(n^2)$, where n is the number of connected clients.

3.2 TWO SERVERS

Our design has been motivated by the spatial locality of players. With this idea in mind, the world can be divided into smaller regions with each region being handled by a different server. Clients can then connect to the proper server handling their region based on the player's position. In our implementation we have created a single world divided into two regions handled by servers S_A and S_B respectively. The regions are rectangular in shape and are transparent to the player, who sees only one continuous world. Every client is connected to at most one server for the most part. The server that the client connects to is determined by the player's location in the world. The client sends its actions to this single server, say S_A, until it reaches a threshold, where it then establishes a connection to the other server, say S_B. This threshold is predetermined based on the view size of the client. While near the edge

of S_A, the client sends its actions to S_A only, but it receives object data from both servers in order to have a complete picture of its environment.

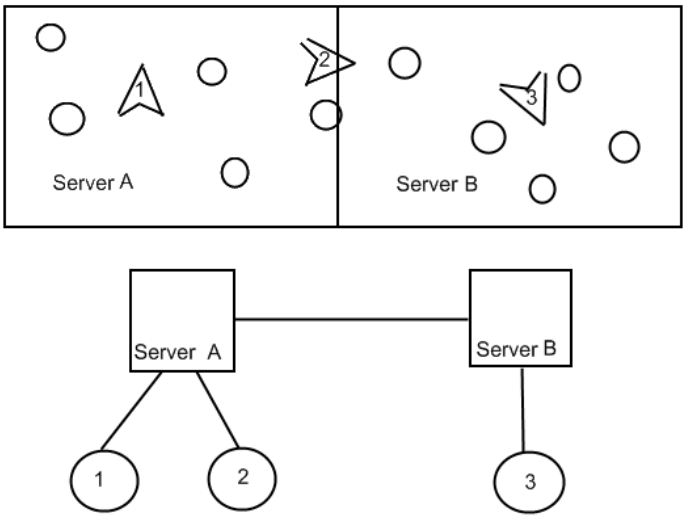

Figure 2: The rectangle represents the server regions, the triangles represent the player's ship, and the circles represent the asteroids. Client 1 receives events from Server A. Client 3 receives events from Server B. Client 2 is currently connected to Server 1. However, due to the player's position near the boundary of Server 1 and Server 2, the client receives events from both Server A and Server B. If the player from Client 2 continues in its current trajectory then the system will then go through the process of disconnecting the client from Server A and connecting it to Server B.

When the player crosses the line between the two servers, the client stops sending its actions to S_A and begins sending the data to S_B. The client maintains a connection with S_A and continues to receive object data from that server until it reaches a threshold where it is well into S_B's region. At this point, the client disconnects from S_A and only interacts with S_B.

Non-player controlled objects are handled in a similar fashion. As an object in S_A enters the threshold between the two servers, S_A sends the object data to S_B. S_B flags the object as an interpolated object from S_A until it crosses the boundary between the servers. S_A then stops sending object data to S_B and flags the object as an interpolated object from S_B, while S_B no longer flags the object as interpolated and begins sending the object data to S_A. S_B continues to send the object data to S_A until the object is well into S_B's region.

4 UNIMPLEMENTED DESIGN IDEAS

As previously mentioned, due to time constraints, we could not produce a complete implementation as we had originally planned. In addition to a two server system, we had produced designs for a larger number of servers in a static configuration as well as a dynamic load balancing configuration. The designs are outlined below, in hopes that they may be implemented in the future.

4.1 STATIC LOAD ON MULTIPLE SERVERS

With a static segmentation of server load, the world is divided up into equal parts for each server to handle. Buffer zones near the edge of each server are set for adjacent servers to be aware of. This allows room for objects and clients to be passed from one server to another. Each server is in communication with up to eight other servers as object data is sent to them: one on each side and one on each corner.

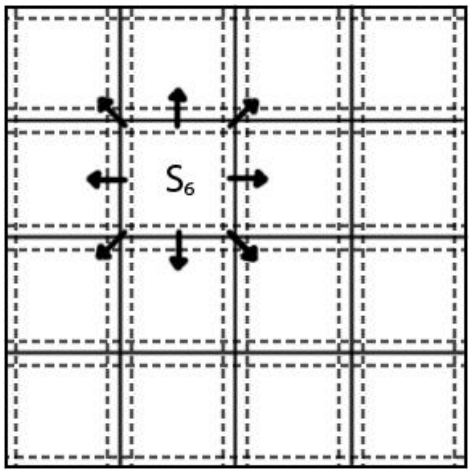

Figure 3: A server sends data from its buffer zone to the respective adjacent servers.

Objects are easily passed between servers. Object data sent from the buffer zone of one server S_1 is interpolated between updates by the adjacent server S_2 receiving the data. S_2 flags the object as an interpolated object as it becomes aware of it but does not take ownership of the object until it leaves S_2's buffer zone and into S_2's main space. Once the object reaches the edge of S_2's buffer zone, the object is no longer flagged as an interpolated object from another server. S_1 relinquishes ownership, since it has completely left its view and S_2 takes complete control.

Clients require additional logic due to the connection exchange that is required. As a client enters a buffer zone of S_1, it will establish a connection with S_2, or if in a corner, the two other adjacent servers as well. Since the server segmentation is static, the client is aware of which server is handling which area as well as its own absolute position in the world. Thus the client can make this decision on its own.

The client will continue to send its input data to S_1, while maintaining ping with S_2. S_2 will begin sending object data as soon as the client reaches the boundary between S_1 and S_2 since the client will require data that will be out of S_1's range from that point. Once the client hits the edge of S_2's buffer zone and into S_2's space, it will begin to send data to S_2 and sever the connection with S_1. S_1 will drop the client from its list of known objects since it has completely left S_1's view, while S_2, having received input data from the client, will know that it now has ownership of the client. In the case that the client never leaves the buffer zone and instead turns around to return to S_1's space, the client will simply disconnect from S_2.

4.2 DYNAMIC LOAD ON MULTIPLE SERVERS

For dynamic load balancing, the server structure consists of a master server, S_m, which takes on the task of distributing loads among slave servers (S_1, S_2…S_8). We assume that the internal network connections between master and slave servers are reliable enough to take on huge traffic load when required. We also assume that there will be an even number of slave servers available at any given time, and the weight of each slave server is the same (same hardware specs, and connection speed). Although our dynamic load design for partitioning of the game world can take weight into consideration with ease, we will not discuss it here to simplify the process.

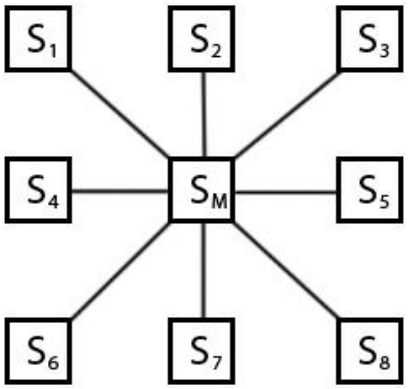

Figure 4: Each square in the graph represents a server, where we have 8 slave servers (S_1, S_2…S_8) for handling clients and one master server for distributing the loads among the slave servers.

The roles of slave servers are very similar to the ones stated in a static setup mentioned in the previous section of the paper. To summarize, their role is to talk to all of the clients connecting to it, all of its neighbouring servers and now the master server as well. As more clients connect to the slave servers, the master server will try to get a snapshot of the universe on a regular interval of *n* minutes (as a function of the total number of clients connected at any given time) by asking all the slave servers to send their region data to calculate the new distribution between slave servers. The distribution between slave servers is calculated by determining how many clients are connected to each region. A "hot spot," or a region that is dense with players, will then be separated into two or more servers.

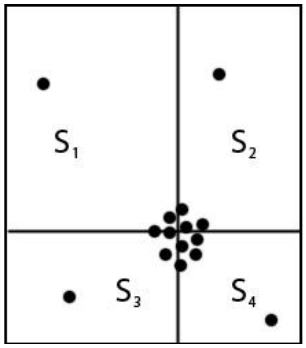

Figure 5: Distribution of workload between four slave servers in order to handle a hot spot.

As mentioned before, server distribution or partition of the game world is done by looking at the number of clients that are connected to each region. Our partition algorithm is very simple. For a six slave server setup we first sort out all of the clients' position by their x-axis, make a horizontal cut, then sort by their y-axis to make two vertical cuts. This simple algorithm will successfully break the load between severs quickly and efficiently, with a small assumption that there will be an even number of slave servers.

With the introduction of dynamic distribution, clients no longer have the knowledge of each slave servers' region coordinate, so it is not possible for clients to make the decision of connecting to the next server when it is making the transition. So our approach is the following:

1) Client is making the transition from S_1 to S_2.
2) S_1 sends the request of connecting to S_2 to the client.
3) The client connects to S_2.
4) The client doesn't send anything except for maintaining ping to S_2.
5) S_1 will continue to send S_2 clients' object data until it crosses line at the edge of the S_2 buffer zone.
6) Once the client crosses the line, S_1 will send a signal to terminate the connection with the client. This will also signify the client to begin sending its data to S_2.

With the slave servers' region changing in an *n* minute interval, clients might have to connect to new servers after the new partition is determined by the master server. With this in mind, the master server needs to send back everything each slave server needs to know in order to run the game properly (meteors, players, and bullets). After all slave servers have received their own new region data, they will issue connection requests with clients within their region if the player did not already exist in the server. However, the process of each slave server sending their snapshot, the master server calculating the partition, then the master server sending back the required objects to the slave servers have to be done in one atomic cycle in order to keep everything in sync. In other words, if we have set our update frequency between slave server and clients to be every 100ms or 10 Hz, then the snapshot, the partitioning, and sending back the objects to each slave servers have to be done under this time frame, or else clients will experience lag after each game world partition update. This might be possible when there are not many slave servers around, but with an increase of the number of slave servers and the game world, this might be difficult to obtain. Thus we advise a second method.

Our second method includes an assumption that the game world starts in a very random state, where everything is spread out. Thus at the beginning, dividing the game world evenly among the slave servers would produce perfect load distribution. This is a reasonable assumption since there should be no clients when the servers initialize, and the clients can be evenly distributed as they begin to connect. As the game progresses, each slave server will send only the number of players in its region to the master server at the same frequency. With number of players at each region given to the master server,

the master server can then adjust the boundaries between servers to slowly redistribute the load. Then only the new region coordinates for each slave servers are sent by the master server. Since the boundaries are gradually changed, objects and clients can hop between servers after a load distribution using the same procedure to transition between servers as outlined above. With only the number of players at each region and the new region coordinates being sent between the master server and the slave servers, our network traffic can be kept to a minimum and it's possible to pass this data around at a 100ms interval. The idea is to let the partitioning of the game world change gradually, thus eliminating the complex step of redistributing objects and clients to slave servers every partition update.

5 LOGICAL IMPLEMENTATION ISSUES

5.1 MAINTAINING GLOBALLY UNIQUE OBJECT IDS

Each object in the game world (the asteroids, bullets, and ships) is assigned a unique ID by the server when the object is created. This is necessary in order for each object data being sent by all entities (servers and clients) to be uniquely identified and properly applied to the correct object. For example, when the server sends the kinematics data for an asteroid, the client needs to know exactly which asteroid that kinematics data belongs to so that it can update the proper asteroid.

In our implementation we use a 32-bit integer as the object ID and manually assign each server a block of IDs. Then each server can simply keep a counter starting at the beginning of its block and increment it to generate a new ID. The main resulting issue is reclaiming IDs of objects which no longer exist (e.g. destroyed asteroids). The game is designed so that objects can be handed off between servers. This means that after a server hands off an object which it has created, the server is then unable to reclaim its ID if that object is destroyed in a portion of the world controlled by a different server. Even if objects are destroyed on the server it was created in, reclaiming IDs would require maintaining an expensive data structure (compared to a simple counter) in order to keep track of available and reclaimed IDs. For example, when an asteroid dies, the server which owns it at that point in time (in other words the server which made the decision that the asteroid is dead) remembers that object ID and then reuses it next time it needs to spawn an object.

One solution is to increase the size of the ID from 32 bits however this would significantly increase bandwidth usage at the server. Our current implementation relies on having a sufficiently large ID space allocated for each server. Once a server runs out of IDs or duplicate IDs are generated, the behaviour of the system is undefined.

5.2 COLLISION DETECTION

Asteroids has many world objects at any one time colliding with each other. As well, different object types interact in different ways when they collide. When asteroids collide with each other, they reverse direction. When a bullet collides with an asteroid both objects get destroyed. When a ship collides with an asteroid the ship gets damaged, and when damaged enough gets destroyed. The world at any given

time could have over a several hundred or thousands of objects in space. It was evident that we needed to implement an efficient collision detection algorithm to alleviate the CPU load on the server side. We used a broad-phase collision detection algorithm, as made popular by the flash game N and the accompanying tutorial [7]. The average running time is O(n) and the worst case running time is O($n$log$n$).

5.3 MAINTAINING CONSISTENCY/SMOOTHNESS

Within our distributed client-server we have servers maintain the state of the game. Each server is responsible for a region of the world and is therefore also responsible for the state of objects in that region. Every 100 milliseconds the client sends the state of the player to the server. This data package we call the kinematics package contains the player's (x, y) coordinates along with the angle of rotation and velocity. The server does the collision detection between objects, updates asteroids and bullets, and sends it to the client. When the client receives the data it reconciles it with its own interpolated data. If the data on the client seems far off from what the server has sent, it throws away its own data and updates it with the received data.

There are three types of world objects in Asteroids: ships, asteroids, and bullets. The server treats all objects the same. All objects move based on their current direction and speed. An object's direction and speed only changes when an event occurs. These events are collisions for all objects, with player input being an additional event for ships. When the server sends a data package to the client, an object data consists of the current (x, y) coordinate, the direction, and the speed of the object. Knowing this, the client can simply perform the same movement calculation that the server does in between updates in order to smooth out the animation. This prevents objects on screen from "jumping" every update cycle.

There was a worry that loss in synchronization or latency could cause jitter in the interpolated animation, but in practise this turned out not to be an issue.

5.4 PLAYER'S AREA OF INTEREST

To optimize bandwidth, the server sends clients only the data of world objects in their area of interest. In Asteroids, the player has a top-down view of their ship and objects around their ship. They do not see objects outside of the scope of this view, which is determined by the screen size of the client. Therefore the screen size is considered the player's area of interest and the player does not see or need to know about objects outside of this area of interest. The server determines what to send to the client by generating a bounding box centered on the player's ship coordinates. The bounding box is slightly bigger than the client's screen size and the server sends all object data that collides with this bounding box. This is calculated when performing our collision detection, so the cost is simply that of adding another object for which to detect collisions on. Our collision detection is already fast with O($n$log$n$) being the worst case and O(n) being the average case.

5.5 DIRTY BIT

A server initially sent all of the objects in the area of interest to each client and other servers. This resulted in a large increase in network traffic as the number of objects and clients grew. Therefore, an optimization was introduced to alleviate the network traffic. Since the clients interpolate the movement of all objects within its view, they do not need to be updated on objects that continue to move in the same direction. The same applies for adjacent servers which can also interpolate objects that they are not in possession of until the next update. Aside from player controlled ships, the only time an object will change course or be removed from the playing field will be during a collision. Since the ships are player controlled, on top of collisions, any input from the player would change the course of the ship as well. Thus, all objects are given a "dirty bit," which indicates whether or not the trajectory of the object has changed since the last successful update to all clients and servers. This has been found to significantly reduce the network traffic as many of the objects in the playing field float in one direction without colliding.

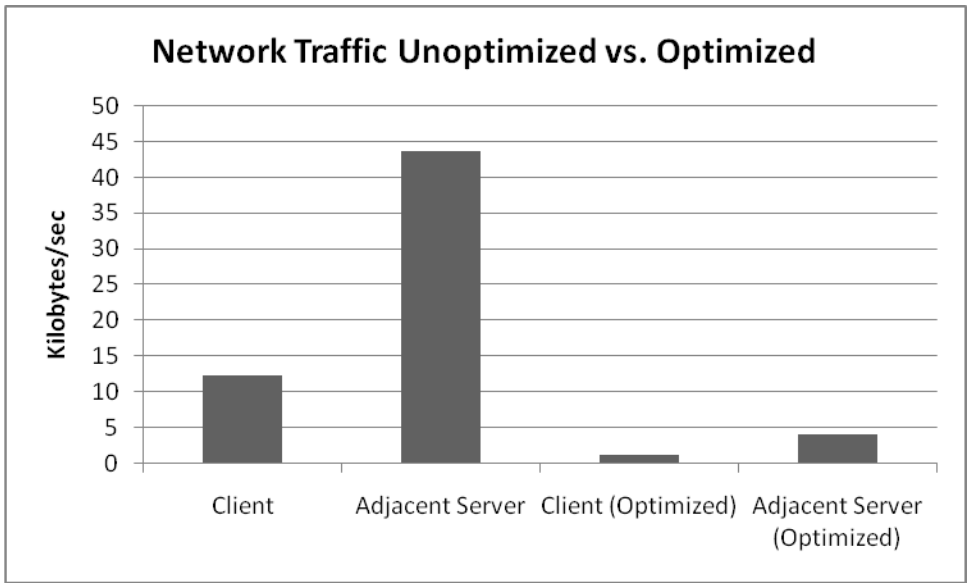

Figure 6: Average server to client (Client) and server to server (Adjacent Server) network traffic comparison with and without the optimization. This test environment contains 1000 objects, two servers, and one client.

In the same vein, clients can reduce network traffic by only sending a minimal ping rather than input data from the player if no new input data is being produced. This would not have a significant effect on the client's upload traffic to the server, but it would allow the server to take advantage of the "dirty bit" which would only be unset to ships once the player stops sending input.

5.6 EVENTS NEAR THE BOUNDARY

Events near the boundary work similarly to a player's area of interest, except here we are dealing with communication between servers. If the boundaries of a region are connected to another region then the servers need to communicate with each so that they can send data to clients with areas of interest that cover two or more regions. Consider two servers A and B that are connected:

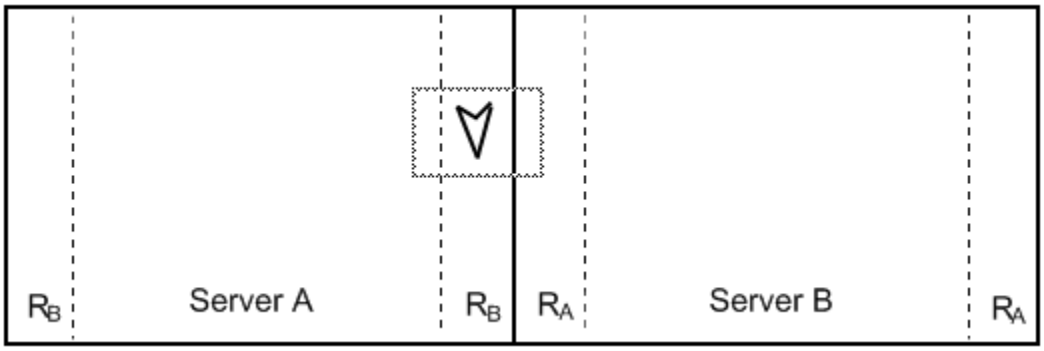

Figure 7: Events near the boundary.

In figure 7, server A needs to know the state of objects on server B in the regions labelled R_A. As seen in the diagram, this is because the client could have an area of interest that requires the state of the object from both servers. Similarly, server B needs to know the state of objects on server A in the regions labelled R_B. Note that our world wraps around. To accomplish this server B simply creates a bounding box for server A. Any objects that are colliding with that bounding box using our collision detection algorithm are sent to server A. Thus server A has knowledge of all objects in regions R_A on server B and server B has knowledge of all objects in regions R_B on server A. Thus server A has all the needed data to be able to send the client the necessary objects in its area of interest even when it is near the boundary between server A and server B.

6 TECHNICAL IMPLEMENTATION ISSUES

We should have coded for network features from the beginning. Initially we developed and completed the client first, then started working on how to integrate the networking component. However, not everything was exactly "network-friendly". For example, we had to isolate routines that dealt with user input, updating, and rendering to better define where and when to receive remote messages. Additionally, we had to define message types and sizes carefully for our objects to optimize for bandwidth which required more refactoring of client side code.

Issues also came up while testing the game, especially because multiplayer games have extra sources of errors. These errors tend to be harder to find and fix in a multiplayer environment. Also prior to working on this project, no team members had worked directly with either of our two key technologies, Microsoft's XNA tools or the Lidgren Networking Library. This resulted in a learning curve period that cut into our implementation time. While XNA was not too difficult to use, Lidgren was a source of some technical problems.

The Lidgren Networking Library manages a pool of buffers. So, when it is needed, the application asks the library for a buffer and the library either gives an already allocated buffer from the pool or creates a new one in memory. Once a buffer has been queued up and sent, the library returns it to the pool. A problem which we ran into was that in our client update loop, we were reusing the same

buffer each iteration for every client instead of allocating a new buffer for each client that needs to be updated. The result was that only the very last client in the list (last to join) would receive any sensible data because an iteration would clobber the buffer intended for the previous client. Basically, it was because we treated the *SendMessage*(*buffer*) function as a synchronous construct (as it would be for a regular socket) while in reality the buffer ends up on a queue instead of getting sent synchronously. This was a tough bug to solve because we initially thought the cause was somewhere in the network. Also the solution was counter intuitive since normally in this situation, only a single buffer is allocated.

7 KEY TECHNOLOGIES USED

Key technologies that we used in our project include the .NET Framework [8] for both the server and client applications. Additionally for the client, we have used Microsoft's XNA 3.0 [2] set of tools which helped us with the development of Asteroids. XNA has been an area of interest for the team and something we all wanted to get our feet wet with. The benefits of using XNA have allowed us to quickly develop the Asteroids client and focus on the server aspect of the project. The content pipeline provided by XNA allowed us to easily load the needed textures for the game and the infrastructure of XNA made the game development easier. As well, we chose C# as our programming language. By using the C# managed environment instead of C/C++, we avoided the common problems of memory leaks. For our programming environment we used Visual Studio 2008 and for source control we used Perforce. XNA provides a networking package on top of .NET sockets but there are a number of limitations like a 31 player limit which made it unsuitable for our purposes. Instead we decided to use the Lidgren Networking Library [3]. It provided us the following packet delivery methods which we used in our project: reliable unordered, unreliable ordered, and reliable ordered which are implemented using single UDP sockets for connecting a client to server. Furthermore, Lidgren also includes several useful features we used such as: message coalescing, bandwidth throttling, local server discovery, connection statistics and simulating bad network conditions.

8 TESTING

Out testing strategy consisted of simulating as large a number of clients as possible connecting to a single instance of the server process. We managed to simulate as many as 400 clients with the server successfully handling the load. Based on the data we obtained, the server could potentially scale to a larger number. The clients used for testing purposes were a stripped down, non-graphical version of the regular client application which only simulate random movement by the player and send the update to the server. Each client consisted of its own thread all running within one process.

The first two datasets were generated by running the server and client processes on the same host. The specification of the testing machine was as follows:

- CPU: Quad-core @3.0GHz, 8MB L2 cache

- RAM: 4GB (memory usage was insignificant, 100MB combined for both processes)
- OS: Windows Server 2008 x64

The number of clients was gradually increased until CPU utilization reached 100%. At this point, 400 clients were connected and the server was utilizing around 30-40% of total CPU time with the client process using the rest.

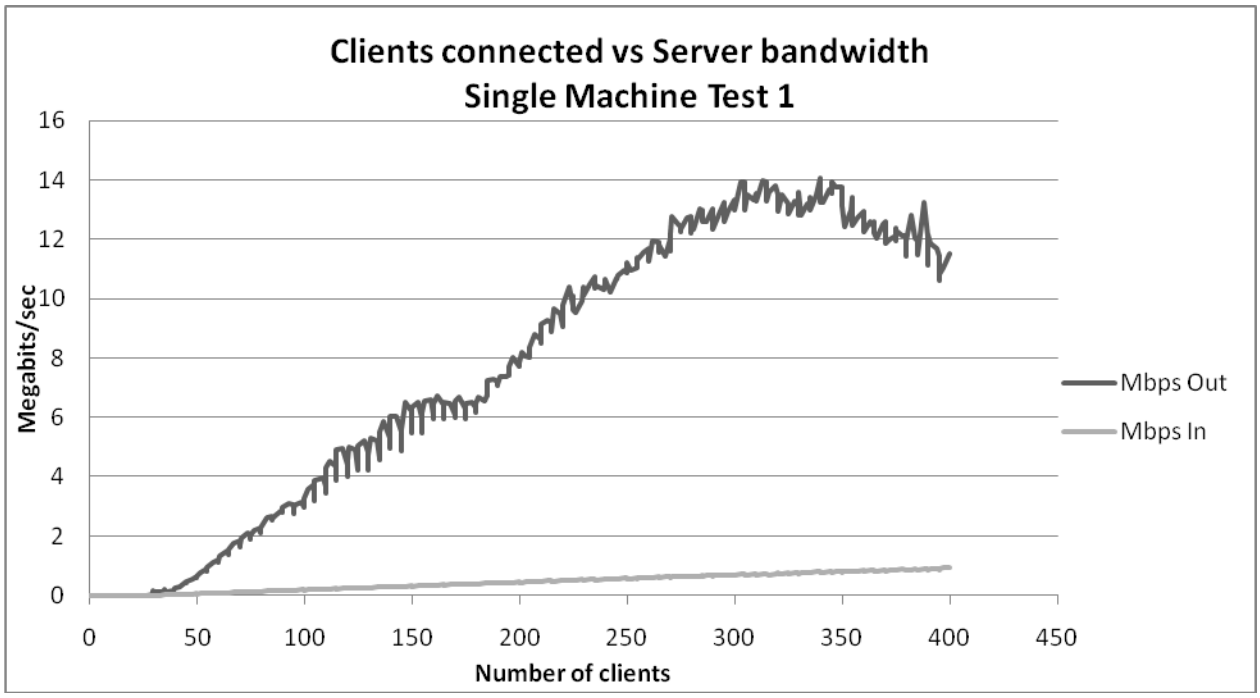

Figure 8: Bandwidth measurement of incoming and outgoing network traffic with both the server and client hosted on one machine. 1st test.

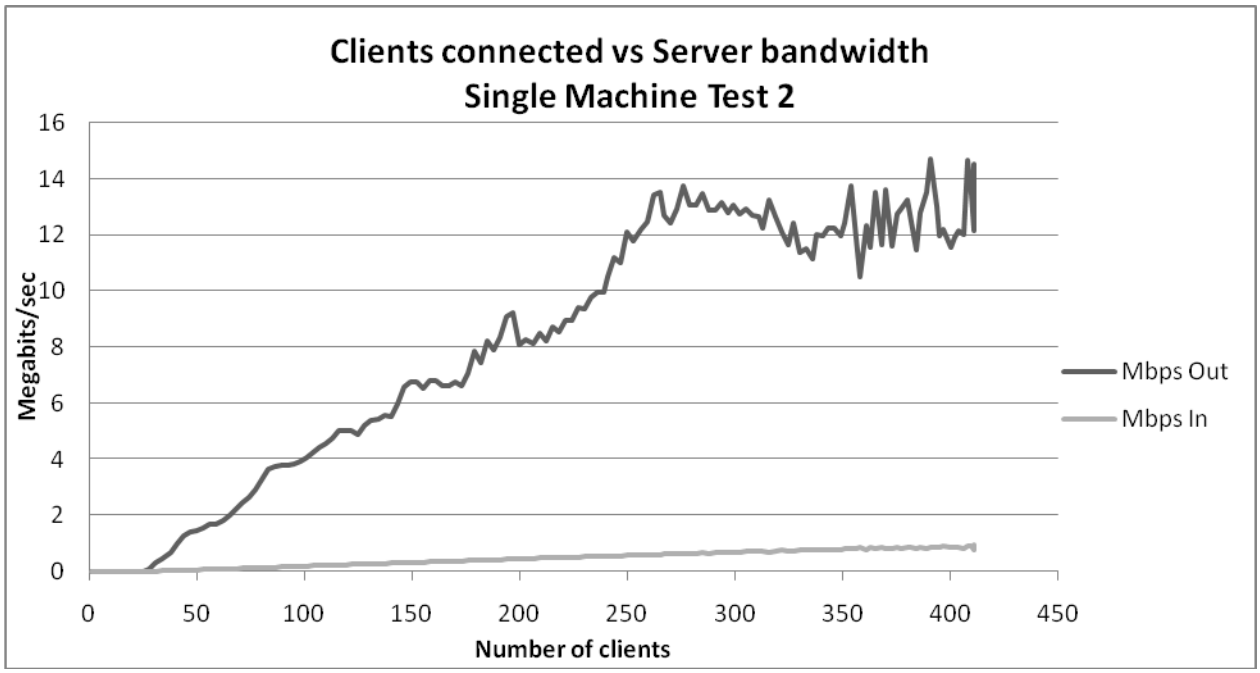

Figure 9: Bandwidth measurement of incoming and outgoing network traffic with both the server and client hosted on one machine. 2nd test.

The bandwidth measurement at the server shows that any bandwidth bottleneck will be on the server's uplink channel. The outgoing bandwidth usage scales $O(n^2)$ in the worst case for n clients, but the average case appears to be linear. Once the number of clients stabilizes the bandwidth usage stays between 10-14 Mbps with the incoming traffic amounting to about 1 Mbps.

The other test setup we tried was with two machines. The quad core machine was used to simulate the clients and the server machine specification was as follows:

- CPU: Pentium 4 @ 2.8GHz, 512KB L2 cache
- RAM: 1GB (again not a factor)
- OS: FreeBSD x86 with Mono 1.2.1*

*Mono is an open source .NET runtime available for *nix systems.

The network link was a 54Mbps wireless connection (the maximum usable bandwidth is much less), which turned out to be the bottleneck. This simulation achieved a mere 100 clients before some began timing out due to network congestion. The outgoing bandwidth usage at the server peaked at 4 Mbps and scaling behaviour was on par with the previous tests. This server's CPU utilization reached about 44% with 100 clients so we expect the machine would have maxed out with around 250 clients, which is less than in the previous test.

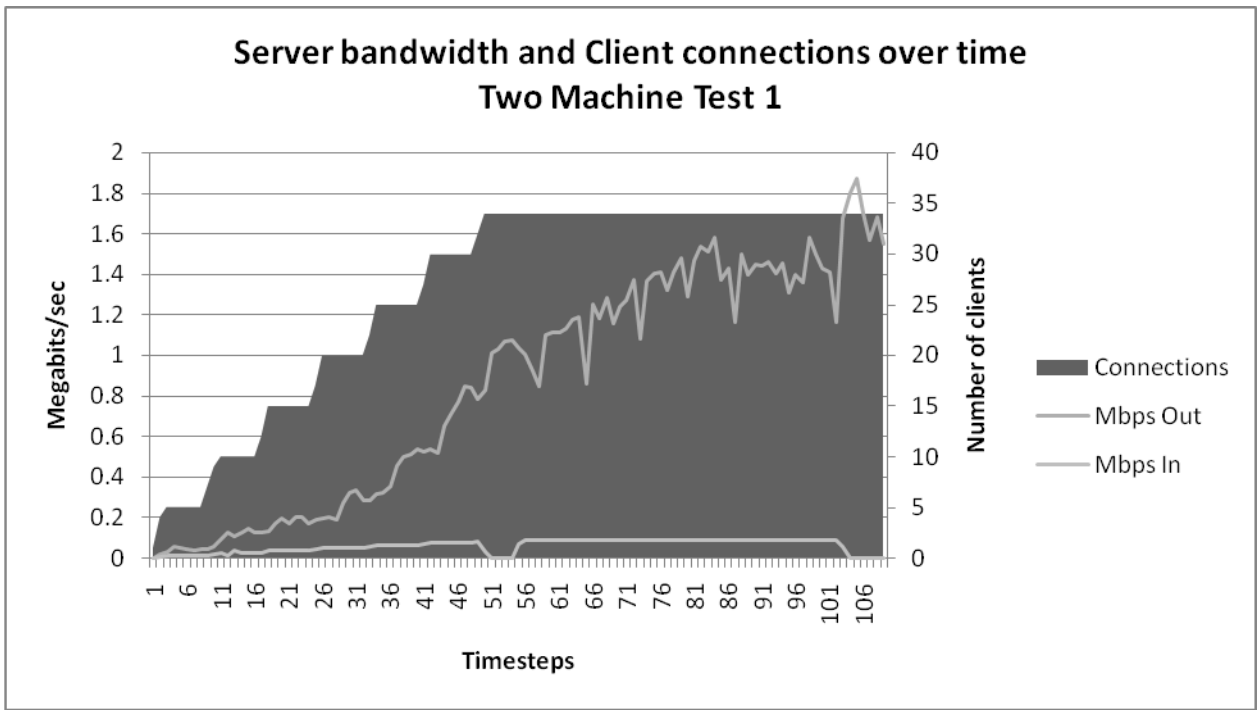

Figure 10: Bandwidth measurement of incoming and outgoing network traffic with the server and client hosted on separate machines. 1st test.

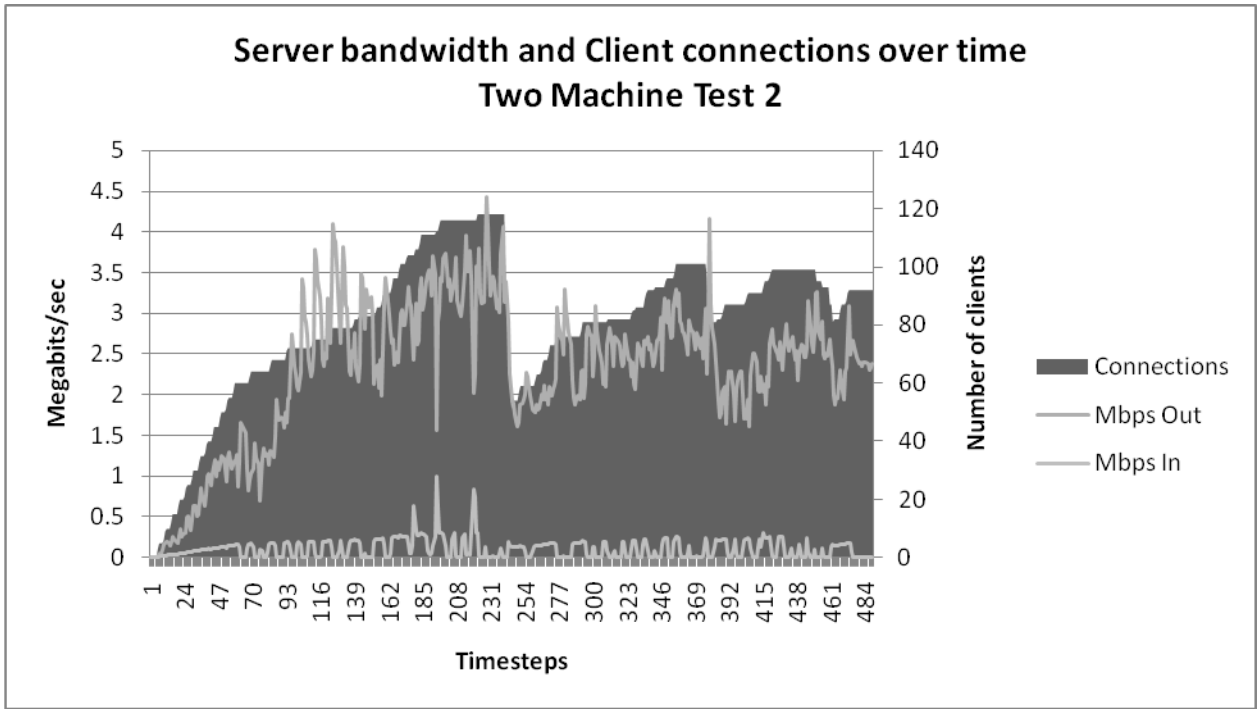

Figure 11: Bandwidth measurement of incoming and outgoing network traffic with the server and client hosted on separate machines. 2nd test.

Notice that the receiving bandwidth on the server sometimes drops to 0. This is indicative of the router dropping packets coming from the clients. The server's outgoing bandwidth never drops because these are statistics are collected by the network library, but it is expected that some of those packets were dropped. Indeed, the machine hosting the clients showed a network usage of 10-20% less than what the server was claiming to send. Because these are UDP packets and the application is treating them as unreliable data, the result would be that the clients would get updates from the server less often but would otherwise be unable to detect any difference. This test simulates the situation where one of the hops in the network between the clients and server becomes congested.

9 LESSONS LEARNED

Implementing a multiplayer game is a lot harder than it seems. A lot of issues needed to be taken into account such as optimizing bandwidth and maintaining consistency for all players in real time. Several important design decisions needed to be made, such as the responsibilities of the client and the server. Also the designs had to include not only how they communicate with each other, but what they should be communicating as well.

XNA is a great tool for streamlining the development of the client. Not only would we recommend it, but Microsoft also has a business model in place for publishing games. However, one drawback that came up when working with XNA in our project was that the executable can only run on other computers that have the XNA redistributable installed. Another drawback was that the XNA library was very heavy as a consequence of abstracting out all of the low level implementation details. As a result, we needed to write a second 'dummy' client without any XNA code for large scale multiplayer testing purposes.

In hindsight, the scope of the project we chose was a bit too large given the limited amount of time we had. As well, we did not anticipate the amount of time it would take to learn to use the tools that we chose to implement the project with. In addition, we spent too much time developing the theories before writing any code, so we had little time to implement and test those ideas.

10 CONCLUSIONS AND FUTURE WORK

In this paper we've discussed our design and implementation of a distributed server architecture for MMO Asteroids. Due to time constraints, we were unable to implement a dynamic load distribution across servers as we had intended, but we have described a method by which such a system may be constructed. Our current implementation consists of two multithreaded servers that host a single game world. However, testing was done with a single multithreaded server, due to the instability of our two server implementation. Our testing showed that the server can scale very well in a modest server environment without a large hit in CPU time or bandwidth. We hope that in the future, we can continue our work to produce a stable implementation of a dynamic load distributed server architecture.

References

[1] Asteroids (video game). http://en.wikipedia.org/wiki/Asteroids_(video_game).

[2] XNA creators club online. http://creators.xna.com/.

[3] Michael Lidgren. lidgren-network. http://code.google.com/p/lidgren-network/.

[4] Sergio Caltagirone , Matthew Keys , Bryan Schlief , Mary Jane Willshire, Architecture for a massively multiplayer online role playing game engine, Journal of Computing Sciences in Colleges, v.18 n.2, p.105-116, December 2002.

[5] Hsiu-Hui Lee , Chin-Hua Sun, Load-balancing for peer-to-peer networked virtual environment, Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games, October 30-31, 2006, Singapore.

[6] Marios Assiotis , Velin Tzanov, A distributed architecture for MMORPG, Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games, October 30-31, 2006, Singapore.

[7] Raigan Burns. N tutorials - Collision detection and response. http://www.harveycartel.org/metanet/tutorials/tutorialA.html.

[8] .NET framework. http://msdn.microsoft.com/netframework/default.aspx.