Minimising Lag in Game Networking Models Without a Central Server

Implementation and Analysis of Client-Hosted and Peer-to-Peer
Networking Models in the Context of Games.

Szymon Jackiewicz
Student Number: 150249751
Supervisor: Dr Graham Morgan

Word Count: ???

MComp Computer Science w Games Engineering
Stage 3

Department of Computer Science
Newcastle University
25 April 2019

Declaration

"I declare that this dissertation represents my own work except, where otherwise stated."

Acknowledgements

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Minimising Lag in Game Networking Models Without a Central Server

Implementation and Analysis of Client-Hosted and Peer-to-Peer Networking Models in the Context of Games

Szymon Jackiewicz

Abstract

A significant share of today's network traffic, consists of game networking. This is due to the emence amount of data that has to fundamentally be sent from client to client (or server) in order to keep a simulation synchronised between several participants. The fundamental principles of this mean that each client should be able to make a change (e.g. movement of their game character), and each other client that participates in the same instance of the simulation should be able to see this change on their local machine. There are several ways of achieving this.

The most common way of achieve a simulation that is synchronised across several users, is to introduce a trusted 3rd party entity that would essentially determine the "Real" state of the simulation. Each client participating in this instance, would submit their changes to this entity which would only implement the changes after checking for validity. The "Real" state would then be sent over to each client so that everyone can update their local simulations to it. Introducing a central entity like this provides a lot of advantages for everyone and is typically implemented by providing central servers that act as the 3rd party entity. The main advantages to this solution include fairness in latency and lag between all players and cheating prevention as all actions are approved before being accepted.

While this solution provides a lot of benefits to the playerbase of a game, it may not always be the best solution. The biggest problem that exists with this solution, is the cost that is associated with the deployment of these servers. Firstly, the cost of renting/running servers that are capable of handling the expected traffic and processing many instances of simulations at the same time could be very high. What makes this issue worse, is that if not enough of these servers are deployed, the experience of all players is compromised through large differences in player ping. This means that servers have to be deployed worldwide which involves further cost.

This downside can be overcome through avoiding the need for a central server through connecting to the other players directly, which introduces several compromises. The different ways of acomplishing a synchronised simulation, without a 3rd party entity, is what I will be investigating in this project through the implementation of 2 different ways of doing this; the Peer-to-Peer Model and the Client-Hosted model. I will be investigating the effectiveness of each solution across several scenarios and aim to come up with a guideline of which one should be used in given situations.

Contents

Chapter 1:   Introduction

Throughout this project, I will be exploring the different methods of providing a synchronised, multiplayer gaming experience on two or more computers on a network. I will be covering the networking basics of how two game clients running on seperate machines can communicate with each other as well as exploring the relevant details of how and why the networking systems in games are designed the way they are.

I will explore the existing methods of writing an online multiplayer game from scratch and aim to provide an implementation of a networking library for online games to communicate without a central server. I also aim to provide analysis of efficieny of different methods of achieving this goal.

1.1   Project Goals

When developing a multiplayer game the most important aspect of the codebase is likely to be the networking functionality and speed. In most games designed for two or more players interacting with each other or the environment, the whole experience can be ruined by slow or inefficient networking implementation. The most common way for games publishers/developers to address this, is to rent out servers space optimised for fast network speeds and a large amount of processing power. This allows each player to connect to a central known entity that can be trusted to be fast enough to handle the load of processing each simulation step and broadcasting information quickly. This solution provides many benefits in terms of security, cheating prevention and fairness, however it is an expensive investment and risk for games developed on a smaller budget.

I will be investigating other options that exist for the smaller budget projects and how to make the gameplay as seamless as possible with the limitations of consumer hardware and network connections. While some aspects of this, such as high ping between clients due to a bad network connection or suboptimal routing are out of control of the implementation, there are many ways that the game netcode can be implemented to minimise latency and provide a synchronised experience for all clients.

Implementation goals:

- Implement a system that allows one client to act as a server for all others. Throughout this document, I will be refering to this as the **Client-Hosted** model.

- Implement a system for each client to send their updates to each other client directly. Throughout this document, I will be refering to this as the **Peer-to-Peer** model.

Data gathering goals:

- TODO

1.2   Basic concepts and terminolory

First of all, I will define some basic concepts and terminolory that will be used throughout this document.

Client: Within the context of this document, a client can be defined as a piece of software responsible to running the game (i.e. game client). A client can also be defined as a computer interacting with a server, however it is possible to run two different instances of a game client on a single machine. This can also be described as a networking endpoint.

Online Multiplayer Game: A video game can be defined as a simulation of a certain scenario that can be manipulated by the player of the game. When talking about online multiplayer games, it can be thought of as a simulation that runs on several clients connected by a network (e.g. LAN or the Internet) that is to be synchronised. When one player performs an action that effects the state of the simulation, this action should also be seen by all participants of this perticular simulation instance and therefore the simulation should remain in the same state across all participating clients.

Ping: In network connections, the ping between different clients refers to the shortest amount of time that is needed for one client to send information to another and receive a response from this client. One client sends a "ICMP echo request" to another networked client (e.g. a game server). The receiving client, then responds with an "ICMP echo reply" back to the original device. The time between sending the request and receiving the reply, is the ping between the two clients.

Lag: The grater the ping between two connected clients, the bigger the difference in the state of each clients' simulation once an action to be synchronised is performed. When a change is made by one client, this change should be seen by other clients participating in the same simulation and lag occurs when this change does not appear instentaneous to the user.

Jitter: The difference in frequency that the messages are sent from a sender and received by the receiver. If a sending client sends packets at a constant rate, they would ideally arrive at the receiver's client at the same rate. This is not always the case however and could lead to some unwanted results in certain applications such as VOIP.

Chapter 2: Background

2.1 Networking principles in games

TODO: talk a bit about UDP, TCP/IP and routing...

2.1.1 Communitaction Issues

In the dissertation *A Network Software Architecture for Large Scale Virtual Environments.*(Macedonia), the author has identified and grouped the most prevelent issues that occur in internet communications. In real time applications such as online games, these issues can be very impactful on the experience of the players.

Data Distribution

Broadcast to each client

Latency

The paper "1500 archers on a 28.8: Network programming in Age of Empires and beyond"(Bettner and Terrano), documents the architecture and implementation of networking approaches in the RTS (Real Time Startegy) games "Age of Empires" 1 and 2. The authors talk about an experiment that has been performed with the game netcode where players were interviewed on whether they felt that their game inputs felt responsive at differnet latencies between the players. They have found that when playing with a latency of 250ms or less, the latency was not noticible at all. Latencies between 250ms and 500ms were "very playable". Anything more than 500ms would start becoming noticible however. They have also found that when playing with a noticible latency, the players have naturally developed a "mental expectation of the lag" between the inputs and an action happening. After developing this "game pace", they would enjoy playing the game at a consistent slower response than altering between slow and fast letency.

What can be seen in this example, is that while making the latency between inputs and actions, does make the game more responsive and fun, noticible jitter when receiving messages can also impact the player's experience in significant ways.

Reliability

UPD can lose packets........ The article "The Internet sucks: Or, what I learned coding X-Wing vs. TIE Fighter"(Lincroft) documents the issues that the developement team encountered when developing the netcode for the 1997 game "X-Wing vs. TIE fighter". The author has experimented with TCP connections in game data transmission and found that "TCP refuses to deliver any of the other packets in the stream while it waits for the next "in order" packet. This is why we would see latencies in the 5-second range." and "if a packet is having a tough time getting to its destination, TCP will actually stop resending it! The theory is that if packets are being dropped that it's due to congestion.". The features that have been implemented into TCP to make it reliable, end up negatively effecting real-time application data traffic and due to the volume of data to be transmitted, should not be used for time sensitive data.

Bandwidth

Consumer hardware and networking slow (esp upload) Talk about how MTU Works

2.1.2   Inconsistant ping between players

There are many different possible reasons for the lag to the server to vary widely from player to player. Firstly, it is possible that there are not enough servers throughout the world or that they are not spread out evenly enough across all regions. Players who live in geographically more remote locations, are likely to experience higher ping to servers compared to those that live in more densly populated cities due to where the servers are likely to be located. Secondly, there is no way of guaranteeing if the clients are going to be using a wired or a wireless connection to connect to the server. Wireless connections can be much slower and are more likely to introduce other potential problems such as packet loss.

Varying ping between players could lead to a problem of a poor experience for a client with a high ping to the server, as they will receive the updates from the server later than every other player and therefore be at a disadvantage if the game requires real time reactions. Unfortunately under some implementations, this also results in a poor experience for the other players too, who despite having resonable ping to the server, can be shot from behind cover by a laggy player who fired a shot before the cover was reached by them on their version of the game state which is delayed compare to others.

Possible solutions to the variable ping problem

Netcode developers of the most popular games, have tried many different solutions to fix or at least mitigate the issue of widely differing ping to the game server between players. One example of a solution here could be "region locking". This is the idea that only players with equally low ping, can connect to the same game instance on a server. This could be done by providing game servers spread out across as many geographical regions as possible and only allow players to connect to their local one. This presents two main issues however. Firstly, this prevents players in different geographical regions from playing together and therefore serperates the community. Also, the issue of players in remote areas playing together and having a suboptimal experience due to the server lag, has not been addressed. The developers of Battlefield 1 have implemented an interesting solution to this problem and I will discuss this further on.

2.2   Networking Model Options

TODO; dev have choice. all have positives and negatives.

2.2.1   The Centralised Server Model

Most AAA online multiplayer games that are played today, make use of the "centralised server model" for synchronising the simulation state between several clients participating in the same simulation. This means that in an example of a First Person Shooter (FPS), if one player presses the "jump" key, their character will jump on their screen. This input information would then also be sent over to the game server. The server would then send the information that this player has jumped, to all other clients. There is a potential problem here however. Given that the ping between the server and client A is α and between the server and client B is β, the time between client A pressing an input and client B being notified of this input can not be less than $\alpha + \beta$ and due to the limitations of physics $\alpha > 0$ and $\beta > 0$. This means that at any given time the lag experienced between clients A and B will be more than $\alpha + \beta$ when aspects such as server tickrate or network congestion are factored in too.

- ✔ Hardware is likely to be powerful enough to handle the stress of many simulations running simultaneously.
- ✔ High bandwidth connection is likely to be used. Minimises the chance of high latency and network issues.
- ✔ Ping to the server is likely to be similar for all players making the game more fair.
- ✔ No client can see the address of any other client.
- ✔ The developer has a lot of control over what is allowed within the game. This allows for anti-cheating systems that are hard to bypass.

- ✘ Expensive to rent out, or buy and maintain, server space. Letting players rent out servers makes the game more expensive for them.
- ✘ Many servers have to be spread out evenly throughout the world to allow for low latency connections.
- ✘ People living in remote locations may not have low latency access to official servers.

Figure 2.1: The attributes of the central server model

2.2.2   Client Hosted Model

The client hosted model is an example of implementing a networking infrastructure without the need for expensive server rental. One of the main advantages of this solution is the idea that one codebase can be written to work as a client hosted model and this codebase can then be adjusted slightly to work on a central server as well if there ever is a need for this in the future. The basic idea behind this implementation method, is that one of the clients, would act as a server for all other participants alongside also being one of the participants. One of the biggest problems with an implementation like this, is that the player that is hosting the game will have a connection to the server with the ping and latency of 0 which will give them an objective advantage in scenarios where quick reaction times are needed. Another potential concern with an implementation like this, is that the player hosting the game, is likely to have consumer grade, lower-end hardware and network connection. It is also possible that a WiFi connection would be used increasing the chance of packet loss and high latency further. There is also a security concern with

any networking application that, in order to function, has to know the IP address of every player that is connected. This information can be easily used to determine the country or even the city that the player is connecting from. It would also be possible to find out information about their hardware or network connection through how frequently a packet arrives from this player. A difficulty can also arrise if the host player loses connection to the other players as either the game has to finish or a "host migration" would have to take place which is a difficult problem.

- ✔ There are no additional costs to the publisher/developer when releasing the game.
- ✔ Players in remote locations can play together with low latency.
- ✔ The codebase is relatively easily transferable to a central server model if the need arises.

- ✖ The host player has neglegible ping to the server. This could be a large advantage.
- ✖ The host player is likely to use a consumer grade connection increasing the risk of packet loss and high latency
- ✖ The host player could be using WiFi to host the game which could significantly increase the chance of packet loss.
- ✖ The host's hardware may not be powerful enough to calculate each simulation step within an accptable tick rate.
- ✖ If the host player is disconnected mid-game, a host migration will have to take place pausing the game for a few seconds or causing the game to finish unexpectadly.
- ✖ The host player can see the IP address of each other player that they are playing with.
- ✖ Cheating could be easy if the host player sends malicious packets to the clients pretending to be the game server.

Figure 2.2: The attributes of the client hosted model

2.2.3 Peer to Peer Model

The idea behind the peer to peer model, is that if two clients can communicate directly, the latency in the connection between them, could be reduced if where wasn't a server that all messages have to go through. The theory here is that the latency beteen players can be as low as theoretically possible which should result in more responsive gameplay. The use of the peer to peer model is often reserved for applications where only a few clients need to communicate with one another and therefore is a common choice for fighting games and RTS (Real Time Strategy) Games.

For the peer to peer model to work, each client (peer), needs to know the IP address of every other client. This presents a potential security risk as any player in the same instance of the simulation, can use the IP address to determine what country, and sometimes city, each client is connecting from. Another potential problem arrises with how the information about each client is distributed to each client. One potential idea of how this could be done, is to have a central server that manages sessions and matchmaking. Clients that are looking for a session with an open spot, would send a join request message to this known server address. This server would match players up based on factors such as player skill. Once enough players have requested to join to start the simulation, the server would broadcast the IP and Port of each client to each other client. From that point, each client has all the information that it needs to start the simulation execution.

If the goal of using the peer to peer model is to avoid the need for server rental or maintenance, this process could also be done with one of the players hosting the game and having the other players know the address of this hosting client. With this setup, the scenario presented above, would work too.

Since each player is communicating with every other client, there is no central "real state" of the simulation and due to latency in connections, the simmulation from each player's perspective could differ slightly. This also means that each client is responsible for maintaining their own state of the simulation. Another concidence of contacting other clients directly, is that in a randomly selected sample of players in a game that is available worldwide, the latency between each player is likely to vary largely. This can often result in having a slower response when interacting with one player than another who could have a lower latency to you. This could make the game feel unresponsive at times seemingly out of nowhere during gameplay.

- ✅ There are no additional costs to the publisher/developer when releasing the game.
- ✅ Players in remote locations can play together with low latency.
- ✅ There is no concept of host advantage, like what is present with the client hosted model.
- ✅ Host migrations are not a large problem since everyone has the full simulation state.

- ➖ Each client communicates directly with other clients making their connection as efficient as possible in theory.
- ➖ Even though a central "authority" is not needed, one of the peers often has to act as a session host to handle invitations and handshakes.
- ➖ Every client runs it's own simulation and is tasked with keeping it updated with everyone else's.

- ❌ The lack of a central authority (such as a game server) makes cheat prevention difficult.
- ❌ Each player in an instance, can see the IP address of every other player they are playing with.
- ❌ Interacting with two different peers with different latencies, will feel inconsistant for the player.
- ❌ Very high bandwidth usage compared to other models.
- ❌ The amount of update messages that need to be sent, increases as the number of players grows.
- ❌ A player with a poor internet connection or underpowered hardware, will make the game feel less responsive to other players.

Figure 2.3: The attributes of the peer to peer model

Chapter 3:   Related work

Throughout this chapter, I will be investigating how networking is implemented in popular modern AAA games. I will also implement a simple online, multiplayer demo game using different ways of implementing networking in the Unity game engine. The experience of using the options that exist in Unity for network game developement, will be similar to what I will aim to implement at a much lower level with my networking template.

3.1   Networking implementations in games

TODO:

3.1.1   Variable ping system in Battlefield 1

An interesting approach to the issue of variable ping in a game has been implemented by DICE in the game Battelfield 1. Given 2 players; player A with a low ping to the server and player B with a ping of <150ms to the server.

When player B fires at a moving player A, player B's client will perform the check concluding that player A has been hit and this information is sent to the server. The server will then perform it's own checks and if the server agrees that this hit is possible, then it sends the hit confirmation to player B and damage information to player A. This approach is called Clientside-Server Authoritative as while the hit registration is calculated on clientside, the server must still confirm that this is valid.

Concidering another scenario, suppose that player A still has a low ping to the server but player B, now has the ping of >150ms. An icon will appear on player B's UI showing an "aim-lead" indicator. Now when the shot is fired in the same scenario, the hit will not register anymore as the hit registration has switched from Clientside-Server Authoritative to Fully-Server Authoritative, meaning that the check is performed only once the shot information is received by the server.

Whilst this implementation makes the game feel less responsive for players with high ping, it provides a lot more fairness for everyone else and allow for players with different pings play in a more fair way.

3.1.2   Networking in Apex Legends

Apex Legends is a "Battle Royale" game by Respawn Entertainment. Due to the genre of this game, the implementation of networking has been implemented in an interesting way. The premise of the game consists of a starting amount of 60 players, all competing to be the last one alive at the end. An interesting aspect of this, is that while at the start of a match, the server might have to work quite hard to effectively synchronise the simulation for 60 different clients, as more and more players die off and less are left, less information needs to be sent to each of the clients, freeing up some server performance.

Since transferring all the information about 60 different players would most likely exceed the average MTU of a UDP packet, with each update, the server sends 1 smaller packet per each player to each player. This means that when 60 players are in the game, 60 packets are sent to 60 different clients from the same server. This is shown in the Netcode analysis in *Apex Legends Netcode Needs A Lot Of Work*(Battle(non)sense).

An interesting outcome of this, is quite noticible network lag that slowly goes away, as less and less players remain in the game. It is also likely due to this, that Respawn felt the need to implement a "Client Authoritative" aiming model. Meaning that the server will favor what the client sees over it's own view of the simulation (i.e. If the client claims that a shot has hit an enemy, the server is likely to agree). The choise for this implementation could have been made to fix two potential issues; reducing the load on the server by reducing the need to perform extencive checks for each shot (this could be significant if a lot of players are playing), making the game feel more responsive on slower client hardware that may need more time to process up to 60 packets that arrive from the server at each update tick.

3.1.3 Destiny 2's uniquely complicated netcode

In 2015, a Bungie developer gave an interseting talk at GDC (Game Developer's Conference): "Shared World Shooter: Destiny's Networked Mission Architecture"(Truman). The game Destiny 2, has decided to use the peer to peer model to create an "always online" world that players could join and leave freely. This resulted in a netcode that was refered to by as having a "uniquely complicated network topology". In a previous Bungie game with multiplayer networking, Halo Reach, a mixture of a peer to peer model and client hosted model has been used in multiplayer matches. Each player would send information such as weapon fire or movement to all other players but there would also be a physics server running on one of the clients in the simulation. The matches were designed to be relatively short and players where incentivised to remain in the game until the end of each match. This meant that if the player running the physics server were to leave, the gameplay flow would have to be interrupted for all other players as the host migration system allocated a new player to run this server. Fortunately, due to the nature of the game, this happened rarely as players were incentivised to remain till the end of the match. In Destiny 2 however, there are many zones (refered to in the talk as "bubbles") that a players would often join and leave as they move around the environment. This made host migrations much more common and would happen roughly "every 160 seconds". This has prompted Bungie to abandon the client hosted physics server aspect of Destiny 2's netcode and move towards renting cloud server space to host the physics engine. In "private bubbles", however, it became inefficient to communicate with a central server for game physics calls so the old system was used there. In summary, Destiny 2 uses peer to peer in order to send packets directly to other players to make the gameplay feel more responsive than sending messages to the server and then a client but has fixed some issues that the game could have if it was fully peer to peer by implementing elements that use client hosted and central server models too.
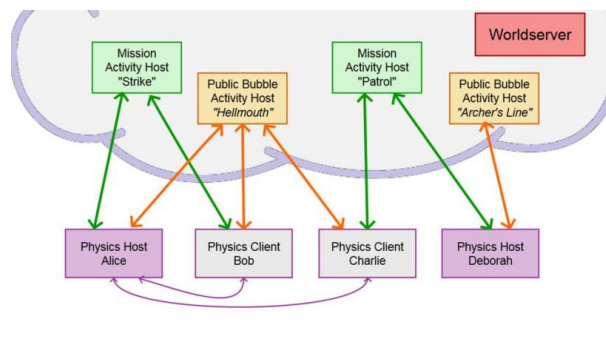

Figure 3.1: Example of how netcode is designed in Destiny 2. Image from the GDC talk: "Shared World Shooter: Destiny's Networked Mission Architecture"(Truman)

Issues with the peer to peer system

3.2   Other networking developement tools

TODO:

3.2.1   Unity's UNET

Allows for easy implementation of Client Hosted model in Unity. Hard to get working like I wanted.

3.2.2   Photon Network in Unity

Similar to UNET, allows for easy and free server sental for central server model

3.2.3   Google's Stadia

At the Game Developers Conference in March 2019, Google has announced a new Game Streaming platform. The talk outlined the plans that would allow developers to develop games for their Linux based platform that would run the game code on Google servers. Users of this service, would be able to play games by sending their inputs to the google servers. All game logic processing would then take place on the Google servers and the audio and video would be returned to the player. This service could potentially allow players to play games on devices with very little processing power, with settings that would normally require much more power.

A potential unexpected advantage of this system, could allow for networking possibilities that would be difficult to make work with current technologies. If a multiplayer game was developed for this system, it could allow for large scale virtual worlds with many players in the same instance as only one simulation would have to be processed for all the players. Each player's perspective would then be sent to them, this wouldn't be much different from how this system would function in a single player scenario.

The largest potential issue with a system like this could be the latency between the player sending an input to the server and then seeing this input reflected in the output. If this latency can be made neglegible through high speed internet connections and high processing power servers, it is possible that a system like this could work. If this latency can be minimised however, this system could prove to be a model that provides the best networking performance for large scale multiplayer experiences.

3.3   Related work summary

TODO: what I am going to focus on and why and this leads nicely into chapter 3

Chapter 4:   Design and Implementation

4.1   Protocols

During any connection that can be deployed between several processes on a machine or even different physical hardware in seperate geographical locaitons, many different protocols come into play. In this section, I will investigate the different methods of transferring data over the Internet and discuss the choices that have to be made in the design of my own protocol to allow for different clients connecting to eachother and sending data.

4.1.1   Protocols in Network Communications

Firstly, in the network layer, most commonly the IP[1] is employed however other options such as X.25 are also available but have more niche uses. These protocols are responsible for "packaging" the data to be sent between two different computers identified by their IP address. The packet from the sending machine, will travel through a network of routers that will eventually lead it to the machine with the IP address of the recieving machine.

Next comes the transport layer where either UDP[2] or TCP[3] can be chosen, both have different properties, advantages, disadvantages, useses and both are used in game networking. The UPD protocol is a simple, connectionless protocol which will simply send a packet from one IP address to another. Since each packet sent with UDP, can take a different route through the router network, there is no guarantee that the packets will be received in the same order as that in which they were sent in. Due to many different reasons, packet loss can also occur. This means that it cannot be guaranteed that a sent packet will even arrive at it's destination at all. Despite these dissadvantages and due to the simplicity of how this protocol was designed with it's connectionless nature, it inately has a major advantage in the speed at which the packets can just be sent out and forgotten about. The TCP protocol, is built upon UDP to add some important features for reliable data sharing at a cost of speed and use in real-time applications. The most important property of TCP includes the assurance that if a packet is not received by a recipient, it is requested to be resent to guarantee that every pacet that is sent, is also recieved. This also means that the packets are aranged in the same order that they were sent meaning that we can be sure that not only data will arrive at the destination, but it will arrive just as we sent it. This implementation has many obvious benefits and in most scenarios, the delay of possibly re-sending a packet, if it was not received, is neglegible. In real-time applications however, this is likely to be an unnecessary waste of time and resources as even if a packet is dropped, resending is likely to not be necessary as by that time, new updated information is available resending the dropped packet less importnat when a packet with new information will be sent anyway. Simply, old information is quickly outdated and it's more important to send new information then old, non-useful information.

The next layer of protocols is the operating system interface or library, that is called

---

[1]IP: Internet Protocol
[2]UDP: User Datagram Protocol
[3]TCP: Transfer Control Protocol

by applications needing to share data using the above protocols. With Windows, a library called "WinSock" is often used, however other options are also available such as enet, asio, RakNet... This is where a programmer would be able to configure which protocols to use (Like TCP or UDP, IP or X.25 amongst many other configurable options).

4.1.2   Connecting the clients together

When connecting multiple clients together, no matter what model is uesd, the session host or the server will need to know what clients will participate in this game session. A brute force solution to this would be to hard code all the information that is needed by each of the participating parties. This solution presents many flaws however. The configuration on each client would have to be configured before the first run on each new participant, as well as when a different connection pattern is wanted. While this solution would be simplest, the issues associated with connecting new clients and synchronising multiple configuration files for the syetem to work together, make this infeasable. A common solution used in the industry is to use something known as a matchmaking server. This is a seperate server with a publicly known IP address, that would be tasked with listening in for join messages from clients and finding the best matches of players based on many different factors such as ping to the game server or player skill. The information about each client would then be passed onto the game server for the processing to start. The matchmaking idea provides many advantages including taking away the computation associated with connecting from the main gameplay server however when developing a networking solution without a central server, this defeats the point. The most appropriate solution that I have found, combines both ideas. Firstly, there will need to be a way for the user to define what the address of the server or session host peer is. This can be done through a configuration file or getting an input from the user. With the server address known, each client can send a message to that address requesting to join a game. The client receiveing this information, would essentially act as a matchmaking server and after getting requests to join from enough players, the game could start.

Design of the matchmaking server

While the basic need for operation of the matchmaking server is slightly different for client hosted and Peer to Peer models, they will also share a lot of logic and functionality. Below is the basic flow of operation for this server.

1. Initialise Networking Library with predefined, known port

2. Listen for "Join Request" messages on the port from clients.

3. If the client is allowed to join the game, reply with "Join Acknowledgement" message.

4.   (a) If the client is a P2P session host
          Broadcast the information about each client to each client.

     (b) If the client is a CH session host
          Send the information about each client to the game server.

   I have experimented with different approaches for implementing this flow. Initially, for simplicity, the implementation consisted only of UDP messages being sent between clients. This means that only one UDP port had to be opened and therefore the same port that is used for the connection logic, would be used for the communication with the game server. After inspecting the solution, I have found that due to the nature of UDP messages, some

aspects of this implementation could be volotile and leave the program in a unexpected state. If a "Join Request" message was not delivered, no acknowledgement message would be received and therefore a timeout waiting system would have to be implemented to resend the join message or the system would be waiting for an acknowledgement for ever. A larger issue could arise however, if the acknowledgement message is not delivered. When the server receives the join message, this client would be added to the server's memory and therefore it would be assumed that the player would be in the game. However, if the player is still waiting for an acknowledgement and the game starts, this player would not know that they are actually in the game. To solve this issue, we could introduce an acknowledgement message for the acknowledgement however as these changes are being made, we are just fixing the intrinsic unreliablity of the UDP protocol. The obvious solution to the issues that have been encountered here, is to use the TCP protocol which exists because it has already fixed the isses addressed here. The revised outcome of the architecture of the matchmaking server, would work in a similar way to the previous design but the TCP protocol would be used. This forces us to implement some changes to the messages that we are expecting to receive from the clients. Firstly, since the set of TCP ports and UDP ports does not overlap (i.e. TCP port 4500 is a completely different port to UDP port 4500), the connection server will need to know what UDP port has been opened by each client. This means that alongside the Join Request message, the clients will need to send the port that the game server will use to send messages. We can still use the source address of the message to know the IP address. With the TCP protocol guaraneeing that no messages are dropped, we can safely assume that each client will get all the information from the server and all clients can proceed to the game loop together. As a result of implementing a TCP server connection, this now means that each client will also have to implement logic for connecting to the server with TCP. Overall, this is a much better solution to the original design as the code for the game logic and connection logic, is more logically organised in seperate classes.

4.1.3   Main Game loop and Broadcasting

The paper *A review on networking and multiplayer computer games*(Smed, Kaukoranta, and Hakonen) discusses the topic of "dead reckoning" in detail. This is the idea of estimating what value is likely to be received from the server in the next packet by evaluating the previous values. In an example of an entity moving through a Virtual Environment at a constant speed in a constant direction and given that the position of this entity is being broadcasted from a game server, the game client can predict that if the previous packets have updated the position of this entity by the same amount every time, it can calculate the next value that will be received before the packet even arrives. This information can be used to interpolate the position of this entity between the packet that has just arrived and what poition it is expected to be in after the next packet arrives, thus giving the client a more smooth simulation. This also makes the smoothness of the simulation less reliant on network quality, though with low quality networking, the issue of "rubber banding"[4] can become prevelent. Even though the implementation of such a system is out of scope of this project, the way the packets are sent from a server to the clients, can make a process like this easier.

The most common implementation for how servers broadcast data from the server to clients in AAA games, is to broadcast all necessary update data at a constant rate, many

---

[4]Rubber Banding is when a client sees an object in a simulation in a certain location but after a update of where this object should be arrives from an authority (server), this position is instantly updated to this new, correct position. From the client's point of view, this looks like the entity has teleported to the new location.

times a second. If a packet is not delivered to one of the clients, there is no need to resend it since a new updated packet will be sent right after the last one. The constant rate at which the packets are sent and the idea that the same amout of 'in-game' time has passed between each consecutive packet that is sent, makes startegies such as dead reckoning easier to implement than if the updates were sent irregularily.

Update rates

When updates are sent to clients over a network, the latency between two machines introduces the physical limitation of the lowest possible time between an action happening on one client and that action being represented on another client somewhere else on the network. This limitation cannot be broken due to the laws of physics and how the information is transmitted through the wires in a network. What makes this delay even larger, is the transmission or update rates of the data from the server to it's clients and vice-versa. When the server broadcasts data 30 times per second, it can be said that it is broadcasting at 30Hz, making the delay between each update roughly 33.33ms. This means that if the server is broadcasting at 60Hz, the delay between a client sending data and another client receiving data should be lower than if the server is broadcasting at 30Hz.

An interesting example of a potential problem that could occur is well explained in *Netcode 101 - What You Need To Know*(Battle(non)sense). In some fast paced games, the update rate of the server can have a large effect on the gameplay, especially in competative scenarios. In many FPS games such as Call of Duty, the gameplay uses guns that fire many times a second. Given an example of a server broadcasing at an update rate of 10Hz and a player using a gun that fires at a rate of 600 RPM (rounds per minute), the time between a player firing two bullets and the time difference between the server sending two updates, is both 100ms. This means that each packet that the server sends will contain information about one bullet. In the same scenario, given that the player is firing a weapon with 750RPM, the delay between two bullets being fired is 80ms. This means that the server will have to send packets that contain up to two bullets. From the point of view of another player that is receiving the damage, it will seem as if one bullet from the gun could deal more damage than another bullet. This is often referenced to as "super bullets". This issue can be solved by increasing the update rate of the server which would also cause packet loss to be less impactful due to the amount of packets that would be received every second.

Tick rates and simulation steps

Given that the higher the update rate from the server to the clients, the better the experience for the players, why not broadcast as many messages every second as possible? The limitation to how many packets can be sent per second, is the speed of how fast the server hardware can calculate each simulation step. Baisic operation of the server can be split into two main concepts: the simulation step calculation and the tick rate.

The term tick rate refers to how many times per second, the game server will process and produce data. At a tick rate of 60Hz, the time between any two ticks would be 16.66ms meaning that the server will have a processing window of that much time to process and broadcast a simulation step. If the server manages to finish the processing of a simulation step within this window (i.e. in this example, processing took less than 16.66ms to complete), it would sleep until the next simulation step needs to be processed. A short processing time on each simulation step means that the clients receive their updates earlier which will lead to reducing the ping between players and make systems

like hit registration feel more responsive in gameplay. When deciding on what tick rate is best to use for the server, it isn't always correct to set the tick rate high as possible. It is also very important to consider that in the most likely worst case scenarios the processing of each simulation step is shorter than the time between ticks. It is paramount that this processing is done within the time window as the server processing not keeping up with the tick rate, often results in inconsistencies such as failing physics or clients rubber banding therefore proving a suboptimal experiance for the clients.

In conclusion, the networking for a game should not be transmitting at a rate that is faster than the average processing time of a single simulation step. To handle this, my project should provide a configurable value of how often the simulation state should be broadcast.

4.1.4   Message Structure Strategies

Data Representation

Depending on the complexity of the game and the amount players participating in the same game instance, it is likely that a large amount of data has to be sent many times from within the main game loop. The data will be transmitted over the internet as a series of bytes and will have to be understood and put together again at the recipient's end. One of the challenges of designing a networking, real-time application, is working around the idea that for any packet sent with UDP, there is no guarantee that it will arrive.

There are many different ways that the same data can be represented and each one is carefully designed to be the most appropriate for it's use. Consider the two figures below of common ways of grouping complex data in an "easily readable" format; XML in Figure 4.1 and JSON in Figure 4.2. These examples have been adapted from the article *Reading and Writing Packets*(Fiedler).

While these formats make it easy for humans to read and understand what is being represented and reading and processing data represented like this is a common operation in software, a lot of characters are used to represent a small amount of data. In both examples, each line would have to be parsed for recognised symbols such as "position" or "health". When a symbol like this is encountered, it's value would have to be found and parsed. The value for position represented as "(0, 0, 0)" would have to be parsed and the numbers extracted to appropriate types in code from chars. The type would have to be checked to make sure that non-numeric characters aren't being represented as numeric types and many other issues such as this arrise. When data has to be read and used many times a second, we would like this process to be as smooth as possible and therefore, we should optimise this representation.

```xml
<world_update world_time="0.0">
  <object id="1" class="player">
    <property name="position" value="(0,0,0)"></property>
    <property name="orientation" value="(1,0,0,0)"></property>
    <property name="velocity" value="(10,0,0)"></property>
    <property name="health" value="100"></property>
    <property name="weapon" value="110"></property>
    ... 100s more properties per-object ...
 </object>
 <object id="110" class="weapon">
   <property type="semi-automatic"></property>
   <property ammo_in_clip="8"></property>
   <property round_in_chamber="true"></property>
 </object>
 ... 1000s more objects ...
</world_update>
```

Figure 4.1: An example of a representation of world data in the XML format

```json
{
  "world_time": 0.0,
  "objects": {
    1: {
      "class": "player",
      "position": "(0,0,0)",
      "orientation": "(1,0,0,0)",
      "velocity": "(10,0,0)",
      "health": 100,
      "weapon": 110
    }
    110: {
      "class": "weapon",
      "type": "semi-automatic"
      "ammo_in_clip": 8,
      "round_in_chamber": 1
    }
    // etc...
  }
}
```

Figure 4.2: An example of a representation of world data in the JSON format

Optimising the data representation

Packet payloads consist of a sequence of bytes. If we were to send over data represented with JSON or XML, each character would be represented as a signed byte (char). These bytes would be processed one by one to extract the useful data. One optimisation that we could do is to represent numerical values like they would be represented in memory, instead of representing them with number characters in base 10.A signed byte can represent values between -128 and 127. An unsigned byte can represent values between 0 and 225. This optimisation would mean that the numerical value is represented in 1 byte rather than up to 4. If we know that the number can not be negative, such as a value representing health, we can use this information to extend the possible range of this value by representing it as an unsigned byte rather than a signed byte. If a number value out of these ranges is to be represented, several concecutive bytes in sequence can be used to represent this value. For example in C++ depending on the architecture and compiler, an `int` type could be represented in 2 or 4 bytes. Another possible optimisation could be done when searching the message for values. Each value represents something in the game and we need a way to distinguish between what each value means. In the examples in Figures 4.1 and 4.2, this is done by having a name for each value. For example the name could be "health" and the value could be "100". An issue with this approach is that multiple bytes have to be read and processed to understand that bytes representing "h", "e", "a" etc. in that order mean that the value will represent health. This process requires many comparisons to check what each value means. There are several ideas that could address this issue. Firstly, instead of a name string, each value could have an ID and that way with 1 byte, it can be determined what the value that follows represents. This would work for up to 256 different values, however if more are needed, a 2 byte ID could be used to represent 65536 diffent values. Comparing 1 or 2 bytes is a much more efficient task than processing a string of undefined length. Another idea could be that each message that is expected from the server is of a predefined format. This could be used by the receving party knowing that, for example, bytes 3 and 4 of the message represent a signed integer representing the x coordinate of the player. If a protocol is designed correctly, it should be possible to read the value at a known location (known offset from where the buffer for the received message starts) when it is needed intead of reading the message to understand what each value is. This approach may not work as well when a lot of data is transmitted however and could even become wasteful if only one value is to be sent but it has to be padded with wasted space in order to offset it to it's correct position.

Detecting and Dealing with Packet Loss

The packet information could contain a certain amount of bits that would be incremented with each simulation step[5]. This counter value could loop round when a maximum value is reached as long as several simulation steps in a row have unique values. The receiver could evaluate this value when received, checking if a packet has been dropped since the last received update. Knowledge about the quality of the connection could be important information when determining how much of the simulation has to be estimated between the received updates and could also be vital information to the player when in a game demanding split-second reaction time allowing them to change their strategy with the knowledge that they may be at a dissadvantage against other players.

The article "The Internet sucks: Or, what I learned coding X-Wing vs. TIE Fighter"(Lincroft),

---

[5]A simulation step refers to a state of values that represent the current state of a simulation. Each time the values are updated, is a new simulation step. This is often done and broadcasted several times a second in central server models.

documents some of the problems that were encountered in the developement of the "X-Wing vs. TIE Fighter" game. The team realised that dropped packets had a significant negative inpact on the game so to counter this, they have implemented a system to resend packets that didn't arrive. This system has initially fixed the issue however in some cases, even the resent packets were getting dropped which coused spikes in bandwidth usage. A solution that they have come up with is that each update packet would also contain a copy of the previous packet attached. This way, even if packet loss occurs, the simulation can stay up to date as long as two or more packets dont get dropped in a row. This solution is quite elegant as Lincroft warns that "if you are using UDP, you shouldn't send small packets", as the headers of UDP packets do not get as compressed as that of TCP packets therefore its more efficient to make use of as many bytes as possible when constructing UDP packets. The solution of including the previous packet data with the current one makes use of this, otherwise wasted, space and could also act as a checksum for the data in the previous packet.

4.2   Message Code Design

Whenever a message is received, the first two bytes will determine the type of this received message. Any unique action to be performed, should have a unique message code associated with it so that it is easy to tell what action needs to be performed after just reading the two byte identifier. If more information needs to be sent along with the message code, this information can be concatenated to the message code byte by byte. The meaning of each byte after the message code will be defined by the message code. The table 4.1 below, contains code descriptions of all possible messages that can be sent and received by my tool.

Understanding message code descriptions:

- The angle brackts (<>) will represent one byte of information (e.g. <ID> signifies one byte number that represents the ID).

- The square brackets ([...]) signify a sequence of undefined length with the pattern that is defined inside them (e.g. [<CHAR>...] signifies that the rest of the message contains a series of charactes each represented as a char(signed byte)).

Message Type	Message Code	Description
Join Request	JR[<NUM_CHAR>...]	Allows a client to send a join request to the server. Up to 5 number chars expected after the message, signifying the UDP port opened for game data.
Join Acknowledgement	JA<ID>	Allows the server to confirm that the client's information has been saved. Is followed by 1 byte indicating the client's ID.
Ping Request	PQ	Message instructing the recipient to reply with PS. Can be used to time the latency in a connection.
Ping Response	RS	This should be sent whenever a PQ message is received.
Update	UP<ID><VAL>	Used by a client to update it's value on the server. Is followed by 1 byte representing the ID of the client and 1 byte representing the new value. If the ID and the address don't match, the update is rejected.
Define	DF<ID>[<CHAR>...]	Used in the peer to peer model to broadcast client information to each client. It is followed by 1 byte representing this client's ID and a char array of undefined length of the format "IP_ADDRESS:PORT". This char array is not terminated by the \0 character.
Current State	CS[<ID><VAL>...]	Used by the server to broadcast it's real state to all clients. When this is received, clients are expected to update their local state to this. It is followed by a non-zero, even amount of bytes representing the client ID and it's value pair.

Table 4.1: Table showing the message codes for distinguishing messages from each other and how each one is expected to be used

4.3 Protocol Flow

4.3.1 Establishing connection

TODO write how code can be reused
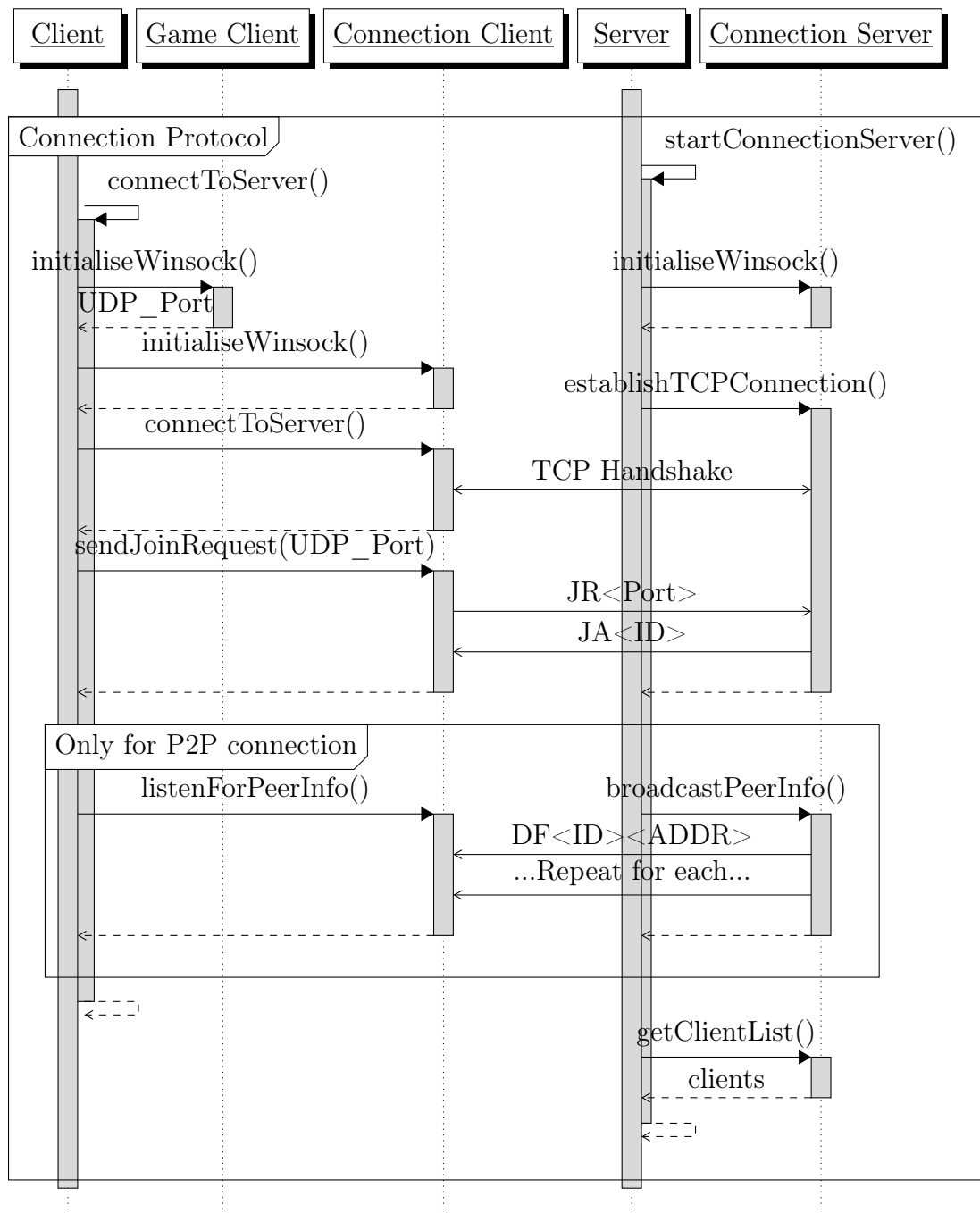
Figure 4.3: Sequence Diagram showing the protocol of any client connecting to a hosting client.

4.3.2   Client and Server communication

TODO used in client hosted. Host client sends to server just like any other, just destination is localhost.
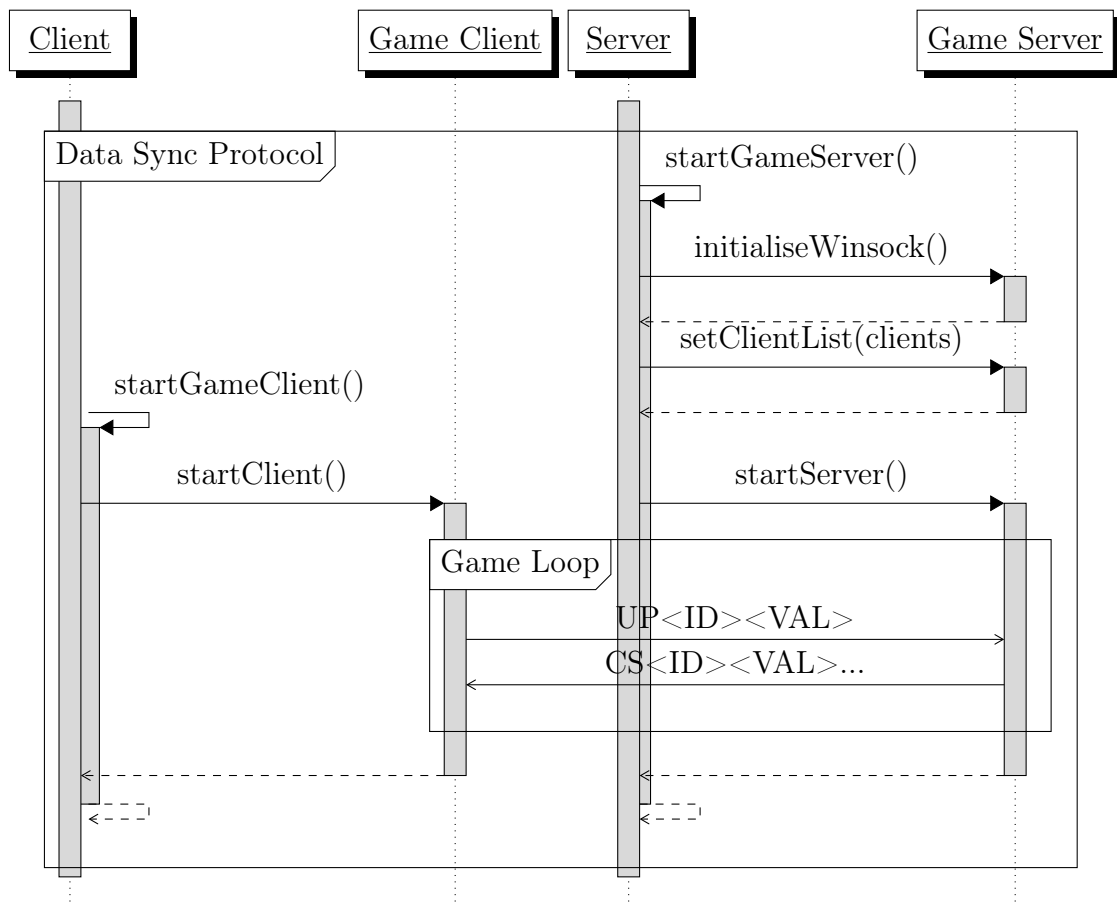
Figure 4.4: Sequence Diagram showing the protocol of sending and receiving updates with the client hosted model.

4.3.3   Peer to Peer data broadcasting

TODO similar to client and server but sends multiple messages instead of one.
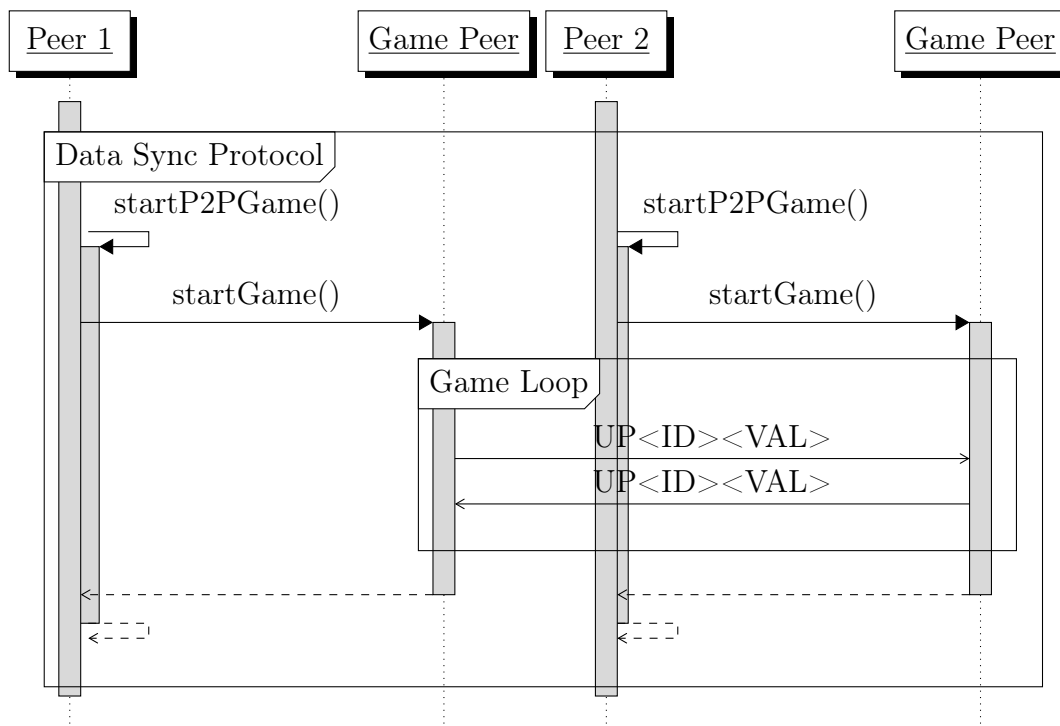
Figure 4.5: Sequence Diagram showing the protocol of sending and receiving updates with the peer to peer model.

4.4   Implementation

I will be writing a simple implementation for the client hosted and peer to peer networking models in c++ using the Winsock library for networking. The goal with this project is to gain insight into how networking systems are implemented in real time applications and identify potential implementation pitfalls for such a system. Alongide the networking logic, I will write a console application that allows me to open several instances of itself to test message transporting between them. This console application will make use of my networking implementations and therefore allow me to easily test my networking functionality as I implement it.

4.4.1   Tools and Technologies

I have identified what technologies I want to use for this project. In this section, I will be analysing each one and provide alternatives that I could have used instead.

Why C++

There are many programming languages avaliable today, each with it's own strengths and weaknesses, each with a purpose that it was designed for. By nature, some langauges are more appropriate for some tasks over others and weighing out the pros and cons is important when making a decision of what language to use for a project.

   Historically, game developement has always fundamentally pushed the boundries of the most powerful hardware that is available at the time of developement. Nowdays, players can enjoy large, expansive virtual worlds with complex mechanics and realistic looking grapics. All these aspects of modern games, demand a large amout of computation that needs to be performed many times a second. The only way to do so much in little time is to make full use of any resources we can use, as efficiently as possible. Originally, the only real way of writing programs, was to manipulate memory locations directly. The management of memory has proved to be a difficult task for any larger scale project, and thus languages such as C and later C++ were born. They allowed us to more easily command the hardware and made programming much more accessable to more people. As time went on, new languages were introduced, that fundamentally did the same thing, but made tasks such as memory management much easier allowing people to quickly write programs and not worry about the intricasies of memory, processing and OS. This developement ended up being a double eadged sword however. Any improvement to ease of use, ended up having a performance cost. Languages such as Java, run in their own virtual machine allowing it to be easily portable to many different operating systems and hardware configurations. This however limits Java to not be able to access hardware resources, such as memory, as easily. The reason why C++, and C to a slightly lessed extent, to remain relevant through these changes in the industry, is that these languages run on the hardware itself rather than a virtual environment. The benefits of this include being able to easily access system memory which is essential when optimising code for performance. C++ has no built in garbage collection that is happening behind the scenes in code execution, the progammer controls which memory is being used and which can be released. The "pay for what you use" nature of C++ lets programmers use the processor power for their computation and not worry about anything else stealing it away.

   These are the main reasons why C++ is used in games and for the same reasons, it is a useful language for any action that has to happen in real time, such as game networking. If my library is written in C++, it would allow a game written in C++ to easily and natively integrate with it. Also, writing efficient packet payloads requires bit-level control

and addressing memory locations with pointers, allows me to do this easily. A good networking library should run in the background of an application and not be noticed by the user, unless something goes wrong. The networking service should steal as little performance power from the main game as possible. This can be done with good C++ programming.

Why focus on Windows

Most popular, modern games that are released in today's games industry, are only developed for Windows, there are several reasons for this operating system bias. Firstly, the production of a game is costly and the developement and QA testing on multiple platforms only increases this cost. The business decision for making the most profit, often comes down to developing for the largest userbase, which in this case is Windows. Microsoft has developed many tools, such as DirectX, to entice this behaviour further by making access to video and sound hardware resources easier which only accelerated this trend. For the same reason, if my intention for my application is to be useful for low-level game developement, it would be wise to target the windows platform.

Another advantage of the Windows platform is the system manager that clearly display performance statistics such as CPU and RAM usage as well as the networking bandwidth used by each application.

Why the Winsock library

Winsock is a C++ networking library developed by Microsoft based on the BSD sockets API. There are many similarities between the BSD and Microsoft implementations and most of the functions share a signiture, meaning that a Windows implementation can be easily adapted to work on Unix systems, with little modifications. Since most of my networking and message management logic will be implemented by me, I will only be using Winsock for establishing TCP connections and sending simple packets. These simple requirements, can be satisfied by any networking library so the choice of Winsock was largely arbitrary. An advantages that Winsock has over other libraries however, include very detailed and well written documentation and example library and the fact that it is being maintained by Microsoft, who also maintains Windows. The large collection of examples allowed me to make a quick start with Winsock and the mechanisms for sending and receiving packets were very easily implemented.

4.4.2   Application Specifications

TODO

What should the application be able to do?

1. Allow the developer using the library, to choose whether they wish to use the peer to peer model or the client hosted model.

2. The tick rate of data broadcasting should be configurable.

3. The only requirement to using the functionality is to clone the library in a known location. Only one import should be needed.

4. Every public intreface should belong to the same namespace.

The application will not:

1. Requirement

2. Requirement

3. Requirement

4. Requirement

4.4.3   Interfaces and usage

Interface for the client hosted model connection

Fixtures 4.6 and 4.7 represent how I intend my library to be used to establish a connection between a client and server. The host of the game would be expected to run both the server and client code from the very start. The client on the host's instance is expected to be configured to connect to the server at the location of `127.0.0.1`, meaning that they will be sending the request just like the other clients but to themselves. This implementation will allow for a client to choose to host a game without joining it if that is ever desired. The connection server should be started before any clients attempts to establish a connection with it. Both of these calls will block the current thread until the connection process is completed. No networking activity should be done until this is finished.

```cpp
GNAT::Server* server = new GNAT::Server();

int successful = 0;
successful = server->startConnectionServer();
if (!successful) {
  // Log and abort
}
```

Figure 4.6: Process of opening server to client connections

```cpp
GNAT::Client* client = new GNAT::Client();

int successful = 0;
// Server Details already configured
successful = client->connectToServer();
if (!successful) {
  // Log and abort
}
```

Figure 4.7: Process of client connecting to server

- The client will return a successful, positive error code if the connection to the server was successful, in this case, it eady is ready to send and receive UDP packets with the server once the game starts.

- The server will return a successful, positive error code if a join request has been received from the correct amount of clients and the join acknowledgement successfully sent to each of them. If this is the case, the server is ready to listen for UDP packets from the clients that have joined and broadcast messages to them.

- If a negative value is returned during the connection process from the client or the server, undefined behaviour can occur if the program proceeds.

Interface for the client hosted model game loop

```
// ... after successful connection ...

successful = server->startGameServer();
if (!successful) {
  // Log and abort
}

// Game successfully finished.
```

Figure 4.8: Process of opening server to client connections

```
// ... after successful connection ...

successful = client->startGameClient();
if (!successful) {
  // Log and abort
}

// Game successfully finished.
```

Figure 4.9: Process of client connecting to server

Interface for the peer to peer model connection

Figures 4.8 and 4.9 represent how I intend my library to be used with a game using a peer to peer model. Hosting the game as session host, should automatically add them as one of the clients in the connection process so no other networking thread needs to be started. Both of these calls will block the current thread until the connection process has finished.

```cpp
GNAT::Peer* peer = new GNAT::Peer();

int successful = 0;
successful = peer->openAsSessionHost();
if (!successful) {
  // Log and abort
}
```

Figure 4.10: Process of opening a peer up to connections from other peers

```cpp
GNAT::Peer* peer = new GNAT::Peer();

int successful = 0;
// Session Host Details already configured
successful = peer->connectToSessionHost();
if (!successful) {
  // Log and abort
}
```

Figure 4.11: Process for a peer connecting to peer session host

- The session host will return a successful, positive error code once the expected amout of clients have joined and the info about each client has been broadcast successfully to each client.

- The joining client will return a successful, positive value once it successfully connects to the session host, receives the join acknowledgement and successfully receives client info about each other client.

- If a negative value is returned during the connection process from the hosting or joining peer, undefined behaviour can occur if the program proceeds.

Interface for the peer to peer model game loop

```
// ... after successful connection ...

successful = peer->startP2PGame();
if (!successful) {
  // Log and abort
}

// Game successfully finished.
```

Figure 4.12: Process of starting the listening and sending threads in a peer to peer game

Conceptual Application Flow

4.4.4   Introducing a delay in broadcasing

Figure 4.13: Messages being broadcast with no delay

Figure 4.14: Messages being broadcast with delay

4.5   Testing my implementation

Chapter 5:   Conclusion and Findings

Appendix A: Appendix Title

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

References

Battle(non)sense (2017). *Netcode 101 - What You Need To Know*. URL: `https://www.youtube.com/watch?v=hiHP0N-jMx8&feature=youtu.be`.

— (2019). *Apex Legends Netcode Needs A Lot Of Work*. URL: `https://www.youtube.com/watch?v=9PfFPW9a90w&t=2s`.

Bettner, Paul and Mark Terrano (2001). "1500 archers on a 28.8: Network programming in Age of Empires and beyond". In: *Presented at GDC2001* 2, 30p.

Fiedler, Glenn (2016). *Reading and Writing Packets*. URL: `https://gafferongames.com/post/reading_and_writing_packets/`.

Lincroft, Peter (1999). "The Internet sucks: Or, what I learned coding X-Wing vs. TIE Fighter". In: *Gamasutra, Sep* 3.

Macedonia, Michael R (1995). *A Network Software Architecture for Large Scale Virtual Environments*. Tech. rep. NAVAL POSTGRADUATE SCHOOL MONTEREY CA.

Smed, Jouni, Timo Kaukoranta, and Harri Hakonen (2002). *A review on networking and multiplayer computer games*. Citeseer.

Truman, Justin (2015). "Shared World Shooter: Destiny's Networked Mission Architecture". GDC talk by Bungie.