Analysis on Networking Issues in Server-less, Multiplayer Game Implementations

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: 15,335

MComp Computer Science w Games Engineering
Stage 3

Department of Computer Science
Newcastle University
3 May 2019

Declaration

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

Acknowledgements

I would like to thank this project's supervisor Dr. Graham Morgan, for the help and advice that he has provided throught all statges of this project. His guidance proved to be invaluable. I would also like to thank Laheem Khan for general help.

Analysis on Networking Issues in Server-less, Multiplayer Game Implementations

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

Szymon Jackiewicz

Abstract

There are many different ways of implementing a system for synchronisating two or more simulation instances. Throughout this dissertation, the options that exist for real-time application development without the need for a central server will be investigated. Many different issues may be encountered when such a solution is chosen and this dissertation will discuss these issues and their potential solutions. Examples of netcode from current games is investgated and analysed. Networking tools available in UNITY have also been experimented with through a small side-project. The development project involves creating a C++ library with functionality for server-less networking. The design and implementation process of a networking library is documented and said library is used to test networking issues such as jitter in client hosted connections. The results and analysis of the tests are provided along with the library code. The conclusion also holds ideas for additional work that could be done in the future to further the project.

Contents

Chapter 1: Introduction

This chapter will introduce this dissertation's subject area, present the purpose of the project and introduce the overarching aims and objectives. Some words used in this document may have several meanings therefore definitions for their meaning in this document has also been provided.

1.1 Subject Area

A significant share of today's network traffic, consists of game networking. This is partially due to the immense amount of data that fundamentally, has to be sent from client to client (or server) in order to keep a simulation synchronised between several participants, however there is also an "always online" trend can also be seen throughout many new game releases and internet culture. The general aim of any real-time simulation, such as an online multiplayer video-game, is for any action that is performed by one participant, to be replicated within the simulation of all other participants. For example in an online multiplayer game that allows for movement of a character with a press of a button, if this button is pressed by one player, all players should see this character move in their instances of the game, from their own perspectives. There are several ways in which this can be achieved.

The most common way in which a simulation is synchronised across several users, involves introducing a trusted 3rd party entity, such as a dedicated server, that would essentially determine the "Real" state of the simulation at any given time. Each client participating in a game instance, would submit their changes to this server which would then implement them into it's own state after checking that these changes are valid. The "Real" state would be broadcast periodically to each client so that everyone can update their local simulations to what is received from the server. Introducing a central entity like this provides a lot of advantages for both the players and developers. Some examples of advantages include relative fairness in latency between all players and effective implementation of a cheat prevention system that is possible because the developers control how the server works.

While this solution provides a lot of benefits to the playerbase of a game, it may not always be the best solution. In fact, the cost that is associated with the deployment of these servers can be unjustifiable for smaller projects such as indie titles. Some sources such as "Land of fire: The rise of the tiny MMO"(Zenke) claim that the cost of bringing an MMO to market are close to $50 million and a large majority of this cost would be used for renting server space.

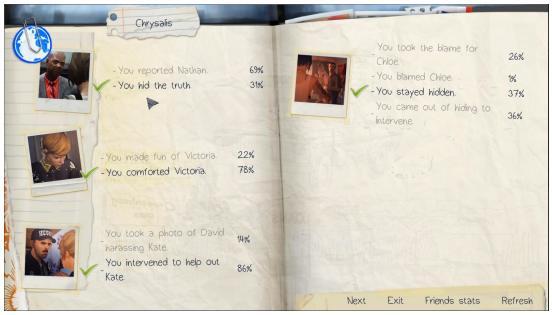
The game servers must be capable of handling a large amount of network traffic and must be powerful enough to run many simulation instances and broadcast the state of each many times a second. What makes this issue worse, is that if not enough of these servers are deployed, the experience of all players is compromised through large differences in player ping. This means that the rented server space not only has to be very powerful, but also available in as many regions of the world as possible.

An alternative for developers who do not have the budget, or find game servers to be a risky investment, is to allow players to connect to each other's game clients directly. Despite introducing some compromises, providing a service that lets players play with

each other directly, may also introduce some new advantages for the players as well as the developers. There are two main methods of achieving a synchronised simulation this way; the peer to peer model and the client hosted model. The focus of this dissertation is to identify the positive and negative attributes of each networking model and to discuss the different issues that are introduced along with techniques to fix them.

1.2 Project Purpose

Most AAA games that are released to the modern game market, have gameplay features that utilise internet functionality. This type of network integration ranges from, giving the player an in-game option and showing them how many players have chosen a given choice (an example of this can be seen below in the game Life is Strange by Square Enix) to multiplayer-only reaction-based competitive gameplay (for example a screenshot from Overwatch by Blizzard can be found below).



(a) Screenshot of choices in Life is Strange. Image from: IGN

(b) Screenshot of Overwatch gameplay. Image from: Game Informer

Games like the examples above are very high-budget AAA projects that are designed and developed by large teams with access to expensive technologies and worldwide servers. The access to the technology and budget that is needed for such large-scale development projects, is simply out of reach of smaller indie developers and therefore most of the popular indie titles are simple, single-player experiences mostly focused on storytelling.

This is also likely due to how fundamentally different the programming approach must be with networking in mind, when compared to a simple single-player, one-instance game. The internet is unpredictable, packets can arrive late or not even arrive at all, what happens when a player loses connection? There are so many new questions that have to be anwsered that for most indie developers, it doesn't seem worthwhile to spend so much time on a networking infrastructure for a game and gamble if there are even going to be enough active players so that they can play together when the game is released. This project exists to investigate just how unpredictable gaming over a network can be and just how should the networking logic be implemented to provide the best experience for the players. The development project will also investigate just how difficult it is to implement real-time networking into a game and the final product could serve as a template for anyone attempting to create a real-time multiplayer experience without the expensive maintenance costs.

1.3 Aims and Objectives

The aim of this dissertation is to compare examples of different methods of implementing server-less, real-time networking. It will also investigate into methods of dealing with common issues that fundamentally arise in computer networking.

These aims can be divided into three separate objectives:

1. Investigate the common practices and techniques used in modern multiplayer games

   There are many different techniques that are employed to counteract the problems that naturally arise when developing any network-enabled application. Potential issues that exist in computer network communications will be identified through the investigation of how these challenges are tackled in popular AAA games.

   The networking models that are used in the industry in these projects, will be analysed for why these choices were made by the developers.

2. Implement the peer to peer and client hosted networking models as a C++ library.

   The focus of this project is on server-less, real-time networking. Two commonly used strategies, that are used in the industry to support this idea, have been identified as; peer to peer and client hosting. Both strategies strive for the same goal, however fundamentally employ different methods of achieving this. The workings of each model's networking logic, will be implemented from scratch so that any configurable variables can be easily controlled and any unnecessary bloat, unrelated to this project's investigation, does not effect the results. From this point onwards, the name used to refer to this project will be the "GNAT (Game Networking API Template)" library.

3. Investigate issues spawned from computer network communication using the implemented networking models in the GNAT library.

   When sending packets during any computer network communication, many things have to be considered. Firstly, any transfer of information can never be instant; there is always some kind of delay. Secondly, without external systems in place, there is no guarantee that any given message packet has arrived at it's intended destination. It is possible that a valid route could not be found or the packet was intercepted. Finally, even if a packet does successfully arrive at it's intended destination, it is impossible to accurately predict the amount of time that this has taken. It is possible that messages sent at a constant rate arrive at the destination sporadically or even out of order to which they were sent in. Any one of these properties of computer network communications can have negative effects on both the application sending and receiving any information. These properties will be analysed for when they can occur and how they can be dealt with.

1.4 Basic concepts and terminolory

First of all, it is important to define some basic concepts and terminology 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 particular simulation instance and therefore the simulation should remain in the same state across all participating clients. Due to the nature of this paper, when discussing games or simulations that use the network, it can be assumed that the given instance would be played by $N > 1$ participants.

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 instantaneous 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

This chapter explores the background knowledge that is associated with every other networking concept explored in this document. The next chapters will build upon the foundation that is established here and will use the understanding of the terminology and concepts to address some issues that are also identified here.

2.1 Networking principles in games

Whenever a message is sent over a computer network, there are many different protocols that are used at different stages of the message sending process. Most data that is sent over the internet, is sent over a protocol stack known as TCP/IP, which includes some useful functionality such as the resending of packets if one does not arrive and the reconstruction of a message on the receiving client's end that makes sure the message arrives just as it was sent. While this process works great for content such as websites, the whole process adds a lot of time on top of the data transmission time of each packet. For data that needs to arrive at it's destination as quickly as possible, a more "bare-bones" protocol is available; UDP. This is a very simple idea of sending the information packet to it's destination and once it is sent, there is nothing else that the sender needs to do. The problem with sending a UDP packet, is that there is no guarantee when, or if, it will arrive at it's destination. There are many different factors that can cause a given data packet to time-out on it's way to the destination or arrive after taking a sub-optimal route there. The following section explains some potential problems that have to be considered when sending information over any computer network.

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 prevalent 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.

Bandwidth

As Macedonia explains, "the available network bandwidth, determines the size and richness of a Virtual Environment". This is in relation to creating a networked virtual environment, however it also applies to any networking operation. The bandwidth of a network is the limit of how much data can travel through it. The bandwidth required grows exponentially as the size of a peer to peer network expands therefore if large simulations are being run on LAN, the bandwidth could become a limiting factor in that scenario.

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 Strategy) 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 different latencies between the players. They have found that when playing with a latency of 250ms or less, the latency was not noticeable at all. Latencies between 250ms and 500ms were "very playable". Anything more than 500ms would start becoming noticeable. They have also found that when playing with a noticeable 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 rather than altering between low and high latency.

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, noticeable jitter when receiving messages can also impact the player's experience in significant ways.

Reliability

When a packet is sent out into the network, there is no guarantee that it will be delivered. To fix this principle, the TCP protocol has been created that builds upon UDP. The article "The Internet sucks: Or, what I learned coding X-Wing vs. TIE Fighter"(Lincroft) documents the issues that the development 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 re-sending 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.

2.1.2 Inconsistent 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 densely 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 reasonable ping to the server, can find the game rules to be unfair towards them. For example in a competitive FPS (First Person Shooter) game that "favours the shooter", a low ping player can be shot from behind cover by a laggy player who fired a shot before the cover was reached on their version of the game state, which is delayed compared to others' versions.

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 separates 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 which will be discussed further on in Section 3.1.1.

2.2 Networking Model Options

There are several different ideas that exist as a solution to the problem of synchronising multiple game clients over a network. The three that are most relevant to this document are the centralised (and distributed) server model, the client hosted model and the peer to peer model. Game netcode developers have to choose one, or multiple, of these implementations when designing their networking logic. Each model has both positive and negative attributes and in the following sections, they will be discussed. The attributes of each model can be found in Appendix A.

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 tick-rate or network congestion are factored in too.

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 arise 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.

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 between 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 arises 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 simulation 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 consequence 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.

The paper "Networking Middleware and Online-Deployment Mechanisms for Java-Based Games"(Carter et al.) provides an example of a game called "Panzer Battalion", where "each peer computes its own game version, no player is having disadvantages because of high network lags". This is an example of a game that uses the peer to peer model to allow for players to play together. The authors also explain that removing a central point of failure, an implementation like this has led to "contradicting game states" which could potentially ruin the game experience.

Chapter 3: Related work

The purpose of this chapter, is to investigate how networking is implemented in popular modern AAA games. To further the research in this topic, an analysis of the development process of using the networking tools available in the Unity game engine will be done.

3.1 Networking implementations in games

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 Battlefield 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.

Considering 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 allows 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 and therefore slow down the sending process, 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 noticeable 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 favour 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 choice for this implementation could have been made to fix two potential issues; reducing the load on the server by reducing the need to perform extensive 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 interesting 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 referred to by Truman 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 (referred to in the talk as "bubbles") that a player 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.

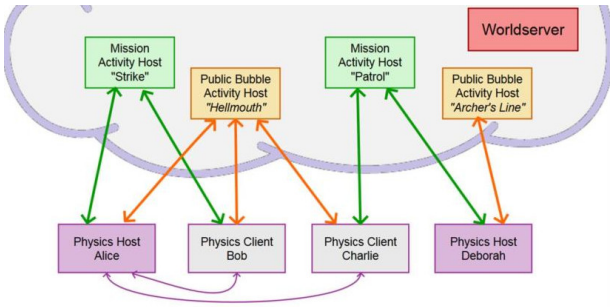


Figure 3.1: Example of how netcode is designed in Destiny 2. Image from the GDC talk: "Shared World Shooter: Destiny's Networked Mission Architecture"(Truman)

3.1.4 Issues with the peer to peer system

The largest issues plaguing the games implemented with the peer to peer system all can be traced back to inconsistencies of state between the peers. This is inevitable when a delay is present in communication between parties. The paper "Networking Middleware and Online-Deployment Mechanisms for Java-Based Games"(Carter et al.) discusses some possible solutions that aim to mitigate or eliminate the idea of desynchronised game state. "In general, during network communication, it can sometimes be assumed that the connection is reliable however we can not assume that messages will arrive in the same order as they are sent".

The first idea that is suggested is called "dead reckoning". The paper *A review on networking and multiplayer computer games*(Smed, Kaukoranta, and Hakonen) discusses this topic 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 another entity, 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 position 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"[1] can become prevalent.

Another idea suggested is called "local lag" which involves the idea of putting incoming packets into a queue before they are used in the game. Using this method, if packets arrive out of order, they can be put in-order in the local buffer. In some game implementations, two actions performed in different order could result in a different outcome (for example rotating a character and moving them forwards). A potential downside that can arise from using this method however, involves the fact that additional delay is incorporated to the process of sending information. This makes use of a system like this in games that require timely reactions potentially degrading for the players' experience. Local lag can be implemented in conjunction with dead reckoning for potentially better results however.

Finally, an option called "time warp" can be utilised for synchronising game state. This technique is often used and well known in the field of Parallel and distributed Discrete Event Simulation (PDES). The paper "Virtual Time"(Jefferson) explains this topic in detail and "Time Warm Mechanism" is well explained. This mechanism listens and applies updates received from other peers when they arrive, however, periodic "snapshots" of the entire game state are saved. If an inconsistency is observed (for example if a straggler message is received from another peer), a "rollback" is applied to reset the simulation to a previous state, before the inconsistency was observed. The system would also have to assume that any messages sent before the rollback was performed, were done so under inaccurate assumptions. If this is the case, the peer can send a "null-message" which would inform the other peers of the inconsistency and lead them to performing their own "rollbacks". This system can often cause "network floods" of "null-messages".

The system of "local lag" assumes that inconsistencies will always occur whereas "time warp" can often be considered to be the "optimistic approach" because it allows each client to calculate it's simulation state assuming no inconsistencies. Both implementations have

[1] 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.

uses in different scenarios.

3.2 Other networking development tools

This section houses the research done that is not related to analysing game netcode. Here, other development tools and platforms that are available to developers are explored. The implementation and usage of tools in the Unity engine will also be explored here and will assist in the design of the GNAT library.

3.2.1 Google's Stadia and future networking options

At the Game Developers Conference in March 2019, Google has announced a new Game Streaming service. The talk outlined the plans that would allow developers to develop games for their Linux based platform and run the game logic on Google's servers. The service would allow users to play games on relatively processor demanding settings on essentially any device with a screen.

The possibilities for multiplayer game systems are a potentially unexpected outcome of this technology. New options could be opened that would be difficult to make work with current technologies, such as a game instance that is shared by an (almost) unbounded amout of players at once. Given that the system works as advertised, running one game instance on the server and just broadcasting each player's perspective on this instance to them could potentially be less demanding on server processing power than the designed use of the platform which would involve processing many different simulations at once.

3.2.2 Networking options in Unity

A game engine has been described as a core of controlling games by providing the main framework and common functions in the paper "Research on key technologies base Unity3D game engine"(Xie). Unity is just one example of many different game engines that are available to a developer, which all aim for the same unified goal but try to achieve it in different ways. Unity is a popular choice for projects ranging from small indie titles to large AAA productions and it is known for being relatively easy to use and prototype with. To investigate the options that are available in the market for online game development, the tools available in the Unity engine have been explored. The following sections detail the experience of using the tools.

A simple 2D game have been implemented which allows the player to fire a projectile, jump and move their character left and right. The goal with this project was to go through the process of transforming a simple, single player game to a simple multiplayer game through the use of Unity's official networking library.

UNET

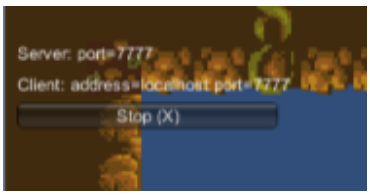
Unity's own UNET library was used for synchronising the state and position of the player. UNET uses the client hosted model that requires one client to act as a game server allowing others to connect to it. The joining clients need to know the IP address of the host in order to connect to them and the host can either join the game themselves or act purely as a server. The figures 3.2, 3.3 and 3.4, show screenshots of the default UI that is provided by the library as well as my game showing the main menu, client and server perspectives respectively. The tile-based pixel art used in the game has been found on the Unity Asset Store under the name "Sunny Land Forest" and has been designed by Ansimuz.

Initially, the implementation of networking features proved to be very simple. A networking manager had to be added to the main scene and the networked scene had to be chosen. Spawn points, and the order that they should be used in, was easy to configure and the UI that allowed the user to connect and play was a simple addition too. Once two different player characters were spawned in, there were some changes that had to be made to the movement scripts that involved specifying different behaviour whether the given character "belongs" to this player, or the other player. Checking if the character belonged to the player was as simple as comparing one boolean. This change can be seen in the change of colour of the character to blue if it is not the player controlled character in figure 3.4.

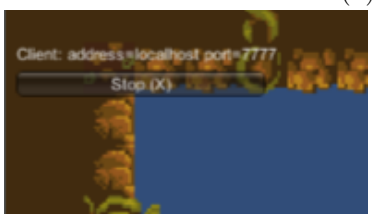
When testing this version of the game, it was found that the movement felt just as smooth inside the multiplayer game session as it did in the single player game. When both the host and the client are playing on localhost (this should in theory provide the lowest latency possible for communications), player 1's movement, would be roughly translated to player 2's perspective correctly, however instead of the movement appearing as smooth as it does for the local player's movement, the enemy character seems to teleport around with every frame appearing in a different location in a radius around where they "should" be at that time. This could be due to the UNET implementation of dead reckoning that does not predict the upcoming positions very well. This teleportation problem occurs for both the host and the client. Another issue appears when a fast-moving projectile is fired. For the host, all projectiles fly just as they should, in a straight line and colliding with a collider. On the client's screen however, it appears as if the tick-rate of the server cannot keep up with the speed of this projectile. Often there are only 1 or 2 frames when the projectile sprite can be seen flashing on the screen in different positions before not being shown again. This appears to happen for both player's projectiles on the client's game. On top of this, there is another issue with this implementation of the game; the desynchronisation of the two players happens very often. After player 2 stops moving, on player 1's screen, player 2's character will continue moving as if a force was being applied in a certain direction on the character collider. This very often results in the two simulations showing completely different states to each other making it hard to know where the other player really is.

All these issues could be fixed by implementing techniques designed for minimising de-sync in peer to peer games and/or sending and reading custom messages between the clients that are designed to work with the game logic, however the default implementation of the official library that is designed to synchronise prefab positions and colliders, appear to work too slowly to be truly useful for quick moving objects, even in the most optimal conditions.

(a) Main Menu UI



(b) Host UI



(c) Client UI

Figure 3.2: UNET default UI

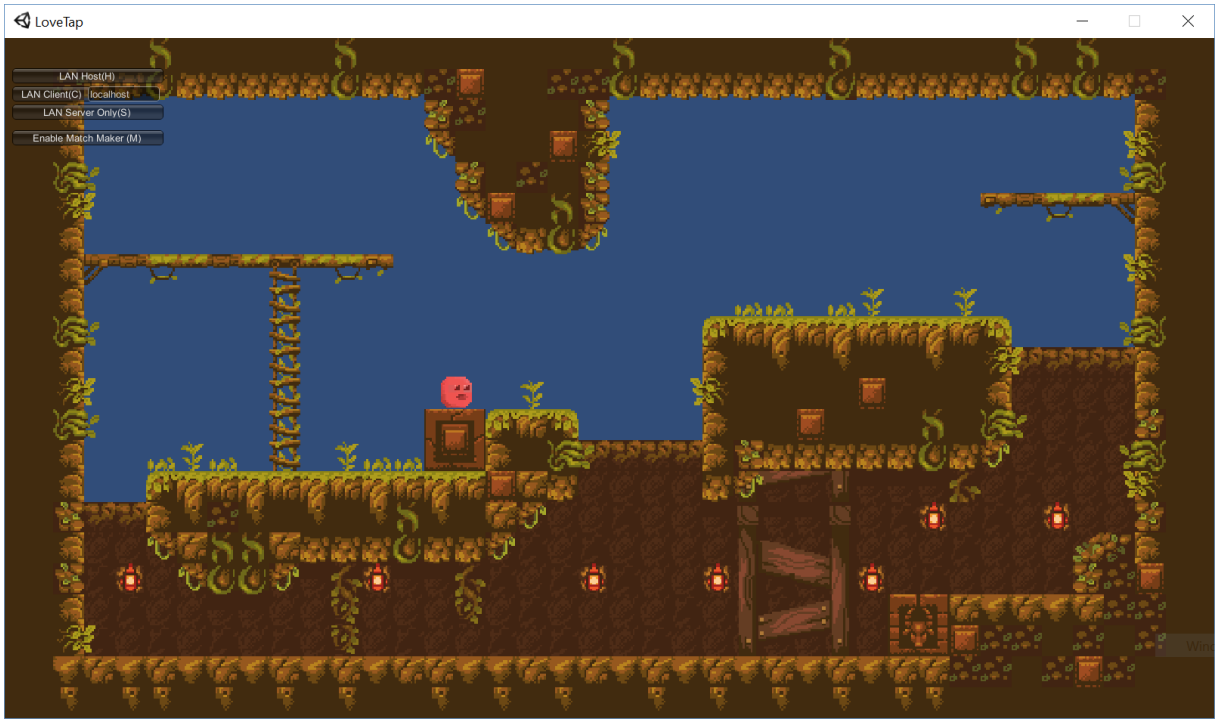
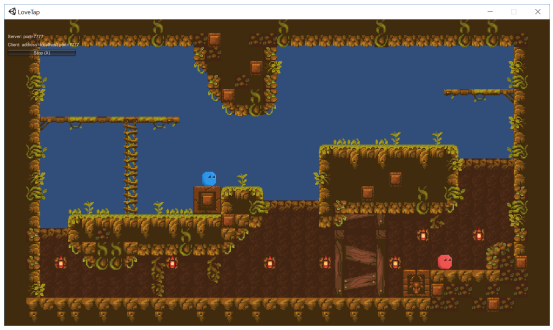


Figure 3.3: Main Menu



(a) Host game screen

(b) Client game screen

Figure 3.4: Main game loop after connection is established

3.3 Related work summary

Most modern high-budget game releases include a competitive multiplayer mode that allow players to challenge their friends or compete with strangers over the internet. The advances in networking architecture allow for seamless multiplayer worlds such as those found in Destiny 2 and the heavily optimised experiences to make the game experience "feel" responsive and fair with systems like those implemented in Battlefield 1 are often taken for granted. In fact, the only time an average consumer is likely to think about how netcode is implemented or the systems that come together to make the game feel fluid, is when a networking system is implemented poorly or a system fails in a given scenario. If the system is implemented well, it is not even noticed or thought about.

This chapter has outlined some potential issues that can be encountered when designing netcode for unusual situations and how some netcode architects have tackled these issues. Further research has been done into how an existing network library works so that some insight from it's design can be used in the design of GNAT. From using UNAT, one may find that through trying to make the library user friendly, the developers have made it hard to customise exactly what information should be synchronised. The library has also made it difficult to send and receive custom messages, which could've been useful to address some of the experienced issues. In summary, UNET has succeeded in making itself easy to integrate into a project and get simple synchronisation working, however it was hard to customise what was synchronised. When designing the GNAT library, both the strengths and weaknesses of the UNAT library should be considered.

Chapter 4: Design

4.1 Protocols

During any connection that can be deployed between several processes on a machine or even different physical hardware in separate geographical locations, many different protocols come into play. In this section, different methods of transferring data over the Internet will be will investigated and the choices made for custom protocols in the GNAT library will be discussed.

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 receiving machine.

Next comes the transport layer where either UDP[2] or TCP[3] can be chosen, both have different properties, advantages, disadvantages, uses 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 disadvantages and due to the simplicity of how this protocol was designed with it's connectionless nature, it innately 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 packet that is sent, is also received. This also means that the packets are arranged 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 negligible. 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 important 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 by applications needing to share data using the above protocols. With Windows, a library

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

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 used, 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 system to work together, make this infeasible. A common solution used in the industry is to use something known as a matchmaking server. This is a separate 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. What could be considered the most appropriate solution, 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 receiving 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.

Several different approaches for implementing this flow have been implemented and experimented with. 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, it turned out that due to the nature of UDP messages, some aspects of this implementation could be

volatile 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 unreliability 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 issues 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 guaranteeing 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 separate classes.

4.1.3 Main Game loop and Broadcasting

A technique that is commonly implemented to smooth out the simulation updates is "dead reckoning" and is explained well in *A review on networking and multiplayer computer games*(Smed, Kaukoranta, and Hakonen). 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 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 amount of 'in-game' time has passed between each consecutive packet that is sent, makes strategies such as dead reckoning easier to implement than if the updates were sent irregularly.

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 competitive 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 broadcasting 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. Basic 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 experience 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, the GNAT library 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 arise. 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.

```
<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

```
{
  "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 consecutive 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 different 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 receiving 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 instead 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[4]. 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 disadvantage against other players.

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

[4]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 development of the "X-Wing vs. TIE Fighter" game. The team realised that dropped packets had a significant negative impact 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 caused 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 do not 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 GNAT.

Understanding message code descriptions:

- The angle brackets (<>) 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 characters 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 recipiant 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

Before a networking application can function, a connection between at least two different parties has to be established so that data can be transferred between them. As mentioned in the Section 4.1.2, the connection process should use the TCP protocol to ensure that all clients have the data that they need. Since this implies that a TCP port will have to be appointed to serve for client connections, it would be wise to separate the connection logic from the game loop and UDP packet logic. Two different classes should be made to handle hosting and joining during the connection process; `ConnectionClient` and `ConnectionServer`. Since the basic information that will be transferred between clients is similar in the peer to peer model and the client hosted model, these classes' logic can be reused for both models. The largest difference between the connection process between the client hosted model and the peer to peer model, is that the peer to peer model has an extra step of broadcasting client information after all clients have received their join acknowledgement message.

Since during the connection process the UDP port number should be sent to the host, any joining client will have to initialise the winsock library for the UDP connection before the join request is sent to the `ConnectionServer`.

After the connection process has finished, the `ConnectionServer` class, will have a list of all the clients that have joined. This list should then be passed onto the class that will run as the server during the main game loop. In the case of the client hosted model server, this list would be passed into `GameServer` and `GameClient` should not have the knowledge of what other players are in the game outside of what is broadcast by the server. In the case of the peer to peer model, the session host will use the list from the `ConnectionServer` whilst each other peer will have to compile their own lists from the information broadcasted at the end of the connection process.

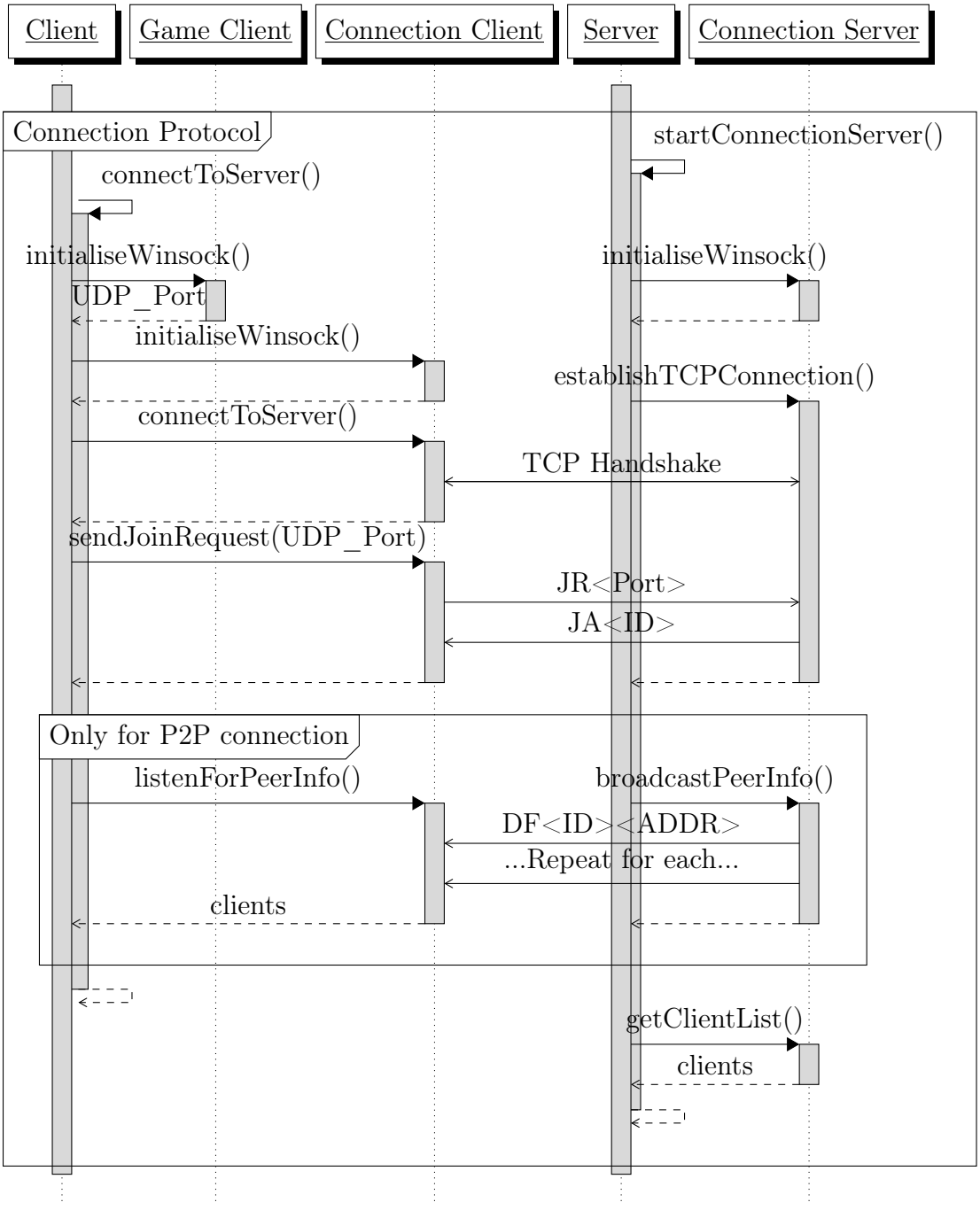


Figure 4.3: Sequence Diagram showing the protocol of any client connecting to a hosting client.

4.3.2 Broadcasting data

The data broadcasting phase happens in the main game loop and this can only begin after the connection has been successfully established between all parties. The Figures 4.4 and 4.5 below, show the process with which data will be sent between parties. The Figures represent the client hosted model and the peer to peer model respectively. In the client hosted model, the port that each client will be connecting to is already known, so the Winsock library can be initialised with this port before the game server data broadcasting begins. This is not the case in the peer to peer model since during the connection phase, each client would have initialised their UDP ports already to connect to the server and therefore this is already done. The main data broadcasting functionality in the game loop will differ slightly between the two models. The clients in the client hosted model, will only ever need to send information to the server so on every tick, only one message is sent. At each tick in the peer to peer model however, each client broadcasts the data to every other client in the game instance. This means that multiple messages have to be sent in the tick when more than 2 players are playing together. Another difference between the two implementations, is that the server broadcasts the public state of the simulation to every client, whilst this is not necessary in the peer to peer system as each update is received directly from the sending client.

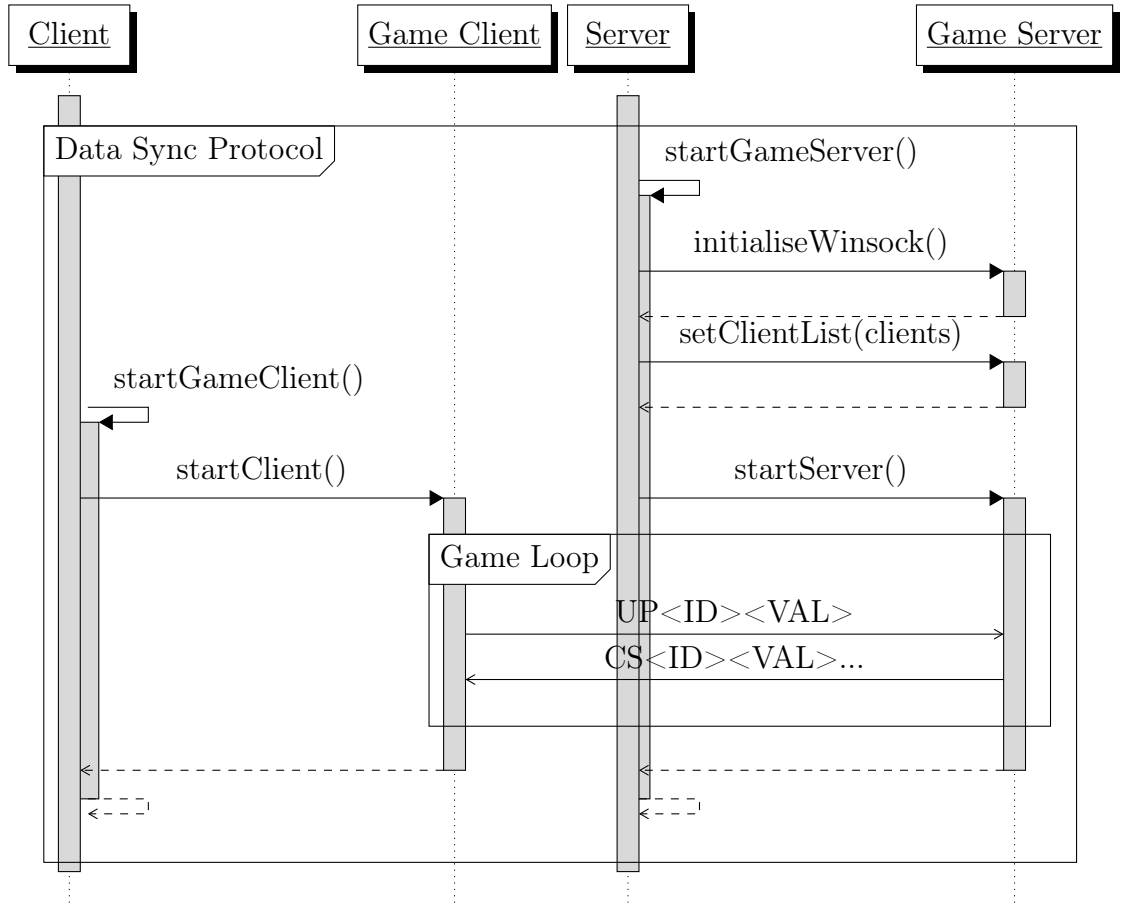


Figure 4.4: Sequence Diagram showing the protocol of sending and receving updates with the client hosted model.

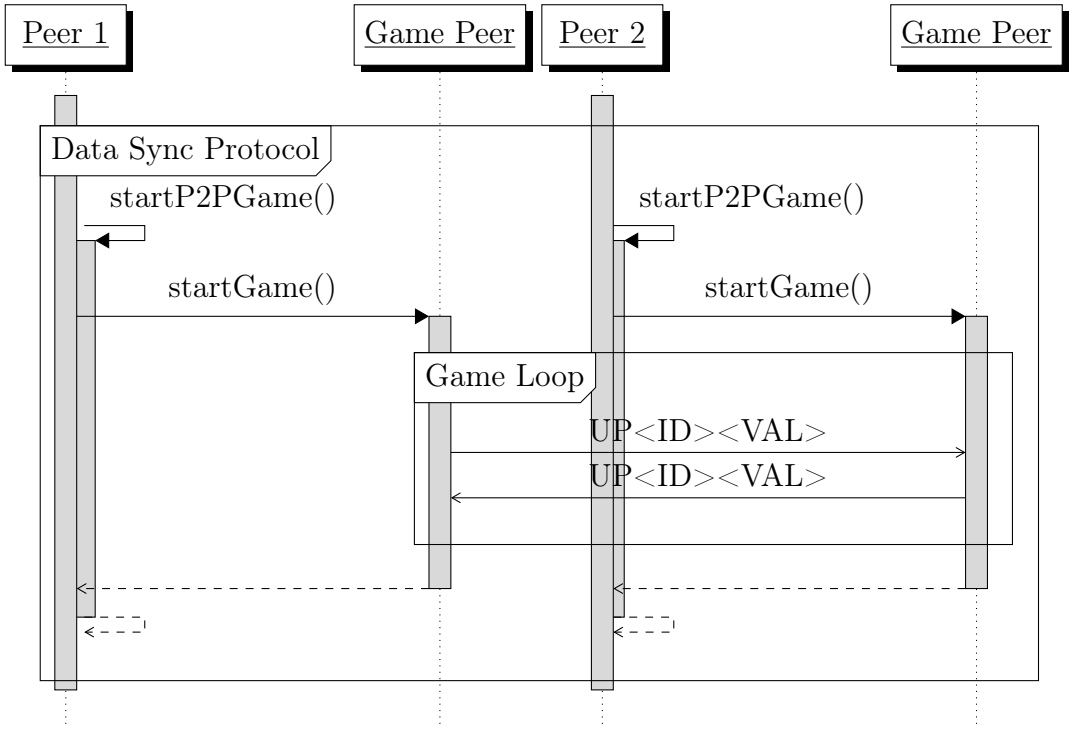


Figure 4.5: Sequence Diagram showing the protocol of sending and receving updates with the peer to peer model.

Chapter 5: Implementation

This chapter documents the implementation for the client hosted and peer to peer networking models in C++ using the Winsock library for the GNAT library. Aside from developing the library, 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. Alongside the networking logic, a simple console application will be produced that will allow users to open several instances of it to test message sending between them. This console application will make use of the GNAT library networking implementations and therefore allow for easy testing of the networking functionality during development.

5.1 Tools and Technologies

Here, the technologies that will be used for this project are identified and analysed.

5.1.1 Why C++

There are many programming languages available today, each with it's own strengths and weaknesses, each with a purpose that it was designed for. By nature, some languages 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 development has always fundamentally pushed the boundaries of the most powerful hardware that is available at the time of development. Nowdays, players can enjoy large, expansive virtual worlds with complex mechanics and realistic looking graphics. All these aspects of modern games, demand a large amount 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 accessible 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 intricacies of memory, processing and OS. This development ended up being a double edged 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 lesser 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 programmer 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 GNAT 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.

5.1.2 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 development 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 the intention for GNAT is to be useful for low-level game development, 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.

5.1.3 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 signature, meaning that a Windows implementation can be easily adapted to work on Unix systems, with little modifications. Since most of the networking and message management logic will be implemented from scratch, Winsock will only be used 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. Examples of advantages that Winsock has over other libraries however, include very detailed and well written documentation and examples as well as the fact that it is being maintained by Microsoft, who also maintains Windows. The large collection of examples allows for a quick start with Winsock and the mechanisms for sending and receiving packets were very easily implemented.

5.2 Application Specifications

Before the start of the development process, it is important to consider what requirements the end product should and should not be able to fulfill. This will outline the scope of the project and determine what will be out of development scope due to time constraints. Alongside the networking code, a commandline application will also be developed. This application will allow the user to issue commands that will call public functions in the GNAT library. This application will allow for easy testing during the development process and will also be used to execute tests on final product.

What should GNAT be able to do?

1. Allow for choice between the peer to peer model or the client hosted model for programs using this code.
   (a) In the client hosted model, the server should broadcast state at a constant pace and listen to update messages from clients. The clients should listen for the server's state broadcast and be able to send update packets.
   (b) In the peer to peer model, each client should be able to broadcast an update message to all clients and listen for update messages from other peers.
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 class that a user would interact with, should belong to a "GNAT" namespace.

What is out of scope for the current implementation of GNAT?

1. Send arbitrary packet payloads. This implementation is designed to test latency and jitter in UDP packets. Dealing with unexpected data and how the data in messages effects the program state is therefore out of scope. It should however be possible to send different values over in the packet payload to be able to differentiate one packet from another.
2. Perform version checks. In a real-world scenario, it is always beneficial to ensure that all clients communicating with another are updated to the same version to ensure compatibility and avoid unexpected state.
3. Implement functionality intended for message jitter mitigation such as "Dead Reckoning". This kind of logic belongs more as part of the game's data processing implementation than a library purely used for sending packets.
4. Functionality for dealing with packet loss such as a cycling packet counter or the previous packet's payload.
5. Implement anti-cheat functionality such as spoofing source IP address of a packet to pretend to be another client.

5.3 Interfaces and usage

The classes that are intended for a user of the GNAT library to instantiate in their code, will have public functions that will perform the networking logic associated with the model in use. In this section, the intended use for these functions is planned out along with their actions and return values.

5.3.1 Error code constants

Whenever possible, each function that has a non-trivial purpose (this does not apply to get and set functions), should return a signed integer value whenever possible. After calling the function, the return value can be checked and from this, it should be possible to determine if the function execution was successful or if and what error has occurred. Generally, if the returned integer is a negative value, the function did not manage to complete what it was supposed to do, or the execution has not completed successfully. This negative value can be compared against known errors that can occur to determine what has happened. This set of expected outcomes will exist in a header file which will contain a different negative value for each error that can happen and a positive value for any other information that should be returned after a successful execution. For performance reasons, these values will be defined as `#define` precompiler statements. An example of implementation of this idea can be seen in figure 5.1.

```
// Startup
#define STARTUP_SUCCESSFUL 0
#define WINSOCK_STARTUP_FAIL -1
#define SOCKET_CREATION_FAIL -2
#define BINDING_SOCKET_FAIL -3
#define GETTING_PORT_FAILED -4
#define GET_ADDR_INFO_FAILED -5

// Client
#define FAILED_TO_CONNECT_TO_SERVER -6
#define NOT_CONNECTED_TO_SERVER -7
#define INVALID_CLIENT_ID -8
#define CONNECTION_ESTABLISHED 0

// Sending Messages
#define FAILED_TO_SEND_MESSAGE -9

// Receiving Messages
#define NO_MESSAGE_TO_RECEIVE 0
#define UNEXPECTED_MESSAGE -10
#define EXEPTION_DURING_MSG_RECEIVE -11

#define UNEXPECTED_PROGRAM_STATE -99

typedef int Error_Code;
```

Figure 5.1: Example code for a header file containing the error messages

5.3.2 Client hosted model connection

Fixtures 5.2 and 5.3 represent how GNAT is intended 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 attempt 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.

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

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

Figure 5.2: Process of opening server to client connections

startConnectionServer() returns:

- A 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.
- A negative error code indicating the error that occured is returned otherwise.

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

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

Figure 5.3: Process of client connecting to server

connectToServer() returns:

- A positive error code value if the connection to the server was successful, the join request was sent successfully and a valid join acknowledgement is received.
- A negative error code value indicating the error that occured otherwise.

5.3.3 Client hosted model game loop

Fixtures 5.4 and 5.5 represent how the GNAT library is intended to be used to start the main game loop responsible for sharing data between server and client. Both will start two threads; one for listening for data and one for sending data. These functions should not be called before a connection has been established between the server and the clients and a non-negative error code returned from the connection functions.

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

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

// Game successfully finished.
```

Figure 5.4: Process of starting the listening and sending threads for the server

startGameServer() returns:

- A positive error code once the game instance has been completed successfully.
- A negative error code if an unexpected error prevents the game server from starting or if the game has finished ungracefully. The error code will differ based on what error has occured so that it can be identified and dealt with.

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

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

// Game successfully finished.
```

Figure 5.5: Process of starting the listening and sending threads for the client

startGameClient() returns:

- A positive error code once the game instance has finished successfully.
- A negative error code can be returned if the game has finished unexpectedly or an error was encountered with the Winsock connections. The error code can be used to identify what error has occured.

5.3.4 Peer to peer model connection

Figures 5.6 and 5.7 represent how the GNAT library is intended 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.

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

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

Figure 5.6: Process of opening a peer up to connections from other peers

openAsSessionHost() returns:

- A positive error code once the expected amout of clients have joined and the info about each client has been broadcast successfully to each client.
- A negative error code will be returned if an unexpected error occurs. An example of this could be a timeout or failure to initialise Winsock on UDP port for game data or TCP port for client connections.

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

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

Figure 5.7: Process for a peer connecting to peer session host

connectToSessionHost() returns:

- A positive value once the client has successfully connected to the session host, received the join acknowledgement and successfully received client information about each other client.
- A negative error code will be returned if an unexpected error occurs. An example of this could be a timeout or failure to initialise Winsock on UDP port for game data or TCP port for client connections. A negative value could also return if invalid data has been received when peer list was expected.

5.3.5 Peer to peer model game loop

Once the connection has been successful with each peer, the game loop should be started by each one. After the connection process, it shouldn't matter if the peer is a session host or just joining a session. In the main game loop, clients will communicate with each other regardless.

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

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

// Game successfully finished.
```

Figure 5.8: Process of starting the listening and sending threads in a peer to peer game

startP2PGame() returns:

- A positive error code once the game instance has been completed successfully.
- A negative error code if not enough client info has been received from the session host or if the game has finished ungracefully. The error code will differ based on what error has occured so that it can be identified and dealt with.

Chapter 6: Evaluation and Findings

6.1 Testing the implementation

In order to keep tests fair, it is important to keep as much of the environment constant as possible between test iterations. Each iteration should have one variable that is changed so that the effects of this change can be easily observed and analysed. The following sections will outline the tests on how changing the volume of traffic that is sent between clients on a consumer network, effects the jitter with which the messages arrive.

The test is set up using the client hosted model where the clients are connected to the network on both an ethernet connection and WiFi. The host is connected to the router over ethernet cables that go through a switch. One client that will connect to the host is started on the same machine as the server; this client will be referred to as the "localhost client". Another client is connected directly to the router and it will be referred to as the "ethernet client". The last client that will connect to the host, is a laptop that is close to the router and connected over WiFi, this will be referred to as the "WiFi client".

Once the connection has been established, the server tick-rate can be configured. This will be the variable that will be changed and the effects of this will be observed at each stage of the experiment. To minimise any external factors that could effect this experiment, all tests were performed at night when the network is under least stress.

6.1.1 Capturing and analysing the results

When writing the GNAT library, one of the most important features was the custom wrapper for a logging library that would go on to be used extensively during the development and testing. This wrapper allows for a message to be written to a log file from anywhere in the program. The library code is written in such a way that any action that is done, is logged under different logging levels depending on the nature of the information. One piece of information that is logged whenever a message is received by a client, is the time that has elapsed since the last message that has been received. This information can then in turn be used to calculate the jitter of incoming messages and this is what is used to generate data in the following experiments.

A simple script was used to extract the values, on each log file (one for each client and therefore three for each test). The script would use RegEx to look for lines where the jitter information is logged and extract the number from the rest of the line. The output would be a file with 1 number per line which can then be fed into another program to generate graphs. The analysis of this data and some examples of the graphs produced in this way, can be found under the section 6.1.2.

In order to compare how varied the numbers were from one another, each result set, was fed as input into a math function calculating the variance (σ^2) of the data. The analysis of this can be found in the section 6.1.3.

6.1.2 Jitter between WiFi and Ethernet connections

The author's expectation for the results from this test were: more anomalous values would be present in WiFi packets rather than Ethernet packets due to the higher chance of packet loss and less reliable connection technology. The jitter in WiFi data is also expected to be more varied than a wired connection. After harvesting and extracting the data from this experiment, over 5000 values were produced for each client and each test. Each number represents the time between two received messages arriving and so it is understandable that if a given packet takes longer then average to reach the client, a larger number will be produced however as a result when the next packet is received (after traveling for what can be assumed to be an average time), it is likely that the next value will be smaller than average. This nature, produces the zig-zag like shape that can be seen in the graphs where if there is a large spike in one direction, it is likely that the next data value, will then produce a large spike in the other direction. The graphs representing this data can be seen in figures 6.1, 6.2 and 6.3. These graphs show the data for the WiFi client and the Ethernet client for 250ms, 64ms and 8ms server tick-rates respectively and each one has been limited to use the same amount of data to make the horizontal axis scale consistent.

For each dataset, the data turned out to be very similar; it can be seen that in general the data for the Ethernet client, arrived within 1 millisecond, however as expected, the data for the WiFi client appeared to be slightly more varied with most of the data arriving within 2 milliseconds of the average. Another anomaly that can be seen with the WiFi connection throughout each test, is that it is much more common for a "random" spike to occur. These spikes could occur due to several reasons. One example is that packet loss occurred causing the wait for the next packet to essentially double. This can be seen in several instances through one value having an abnormally large value but not followed by an abnormally small value directly after.

An effect that can be observed with the different tick-rates, is that it is much less common for abnormally large or small, anomalous data to be seen in WiFi connections as the tick-rate gets faster. As the tick-rate increases, each anomalous result (whether it is due to packet loss or suboptimal routing) is much less impactful if there is another packet arriving soon after. An exception to this rule however can be seen in the 8ms test. The packets in the data for the 8ms test appear much more varied in general more anomalous spikes can be seen in both WiFi and Ethernet data (despite this trend being more obvious in WiFi data). To show these anomalous spikes more clearly, the scale has been stretched to 2.5ms per horizontal line instead of 1ms like it is on the other graphs. Unfortunately, it is unclear why this exception occurs in this test when in every other test, the variance can be seen to be getting smaller.

6.1.3 Calculating the variance in jitter

The next question to answer was: Objectively, which collection of gathered data, had the least variance in it? This data can be seen represented in a bar chart in figure 6.4. The σ^2 values for WiFi at the tick-rate of 500 and 1000 milliseconds, were much larger than the other results due to having to wait for a much longer time if a packet was lost. Even when not considering these anomalous values however, it can be seen that the variance in the receiving rate is on average 30% larger than the average of ethernet and localhost data.

One unexpected result was that even the localhost client experienced a certain level of jitter in all tests. This is likely be due to the operating system prioritising certain programs over others as well how the NIC hardware and winsock manages sending and receiving of data.

Another surprising result was once again seen at the 8ms test. Following on from the previous section, it can be seen that the variance of the data increases when compared to the other results despite what would intuitively be the most "reliable" constant stream of data. The fact that the rise can be seen in all three clients, indicates that there may be some external factor (such as the host's computer's NIC or consumer grade router hardware), that interferes with these results. In general however, it is unlikely to see broadcast rates higher than 63Hz (around 16ms delay) anyway.

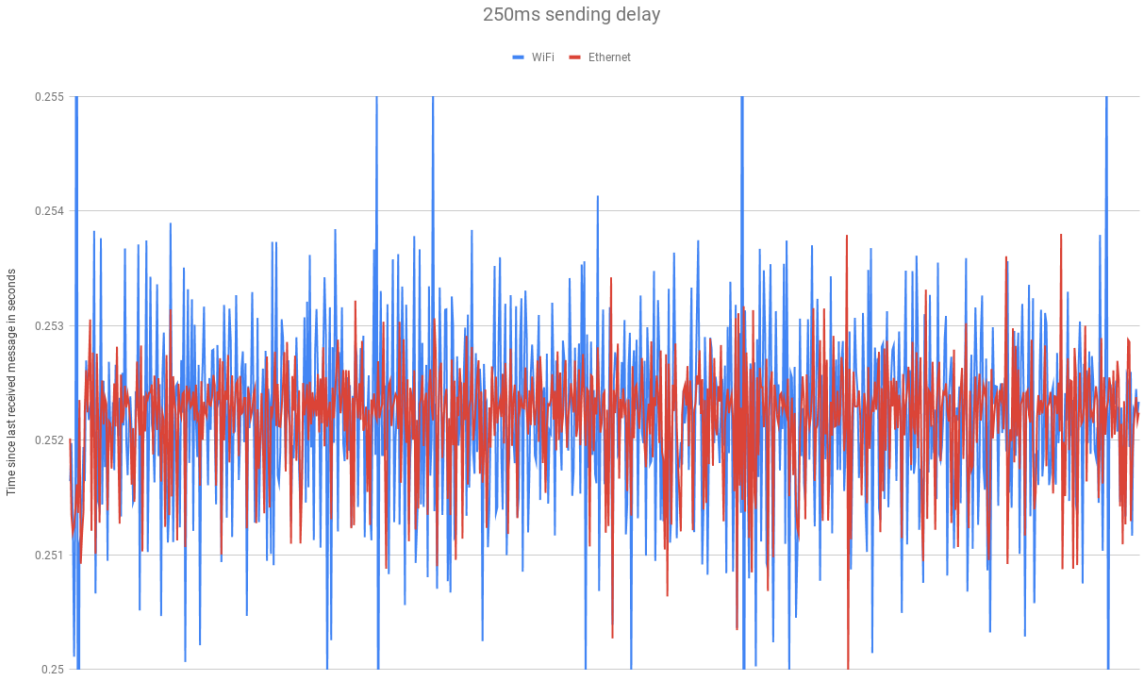


Figure 6.1: 250ms Delay

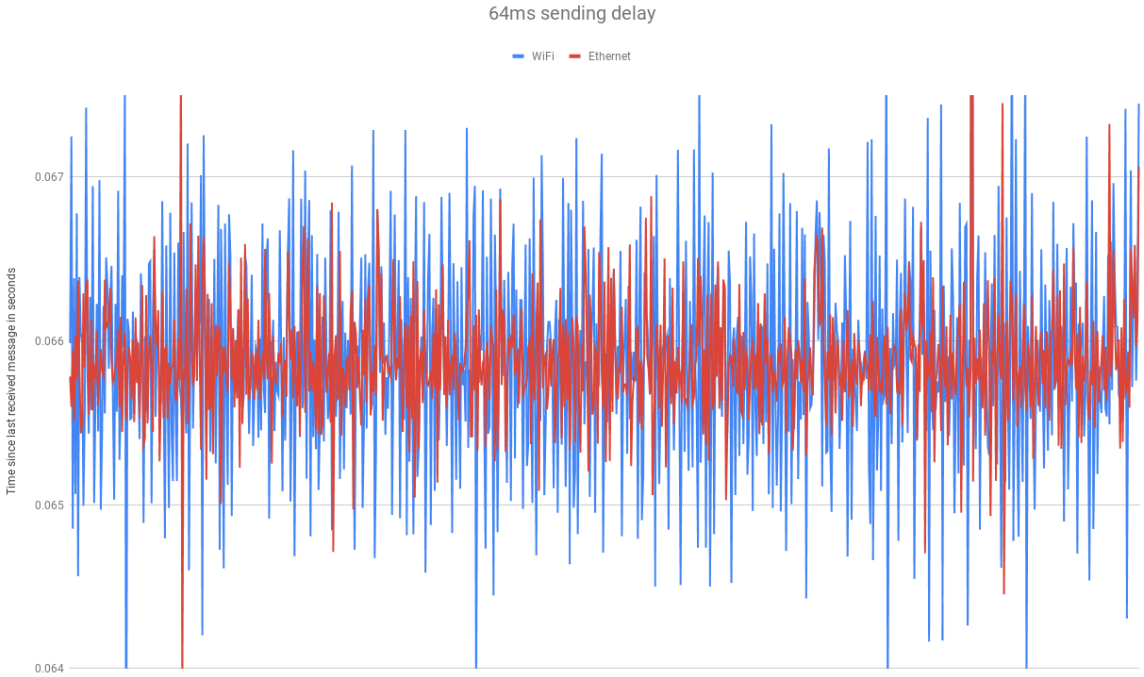


Figure 6.2: 64ms Delay

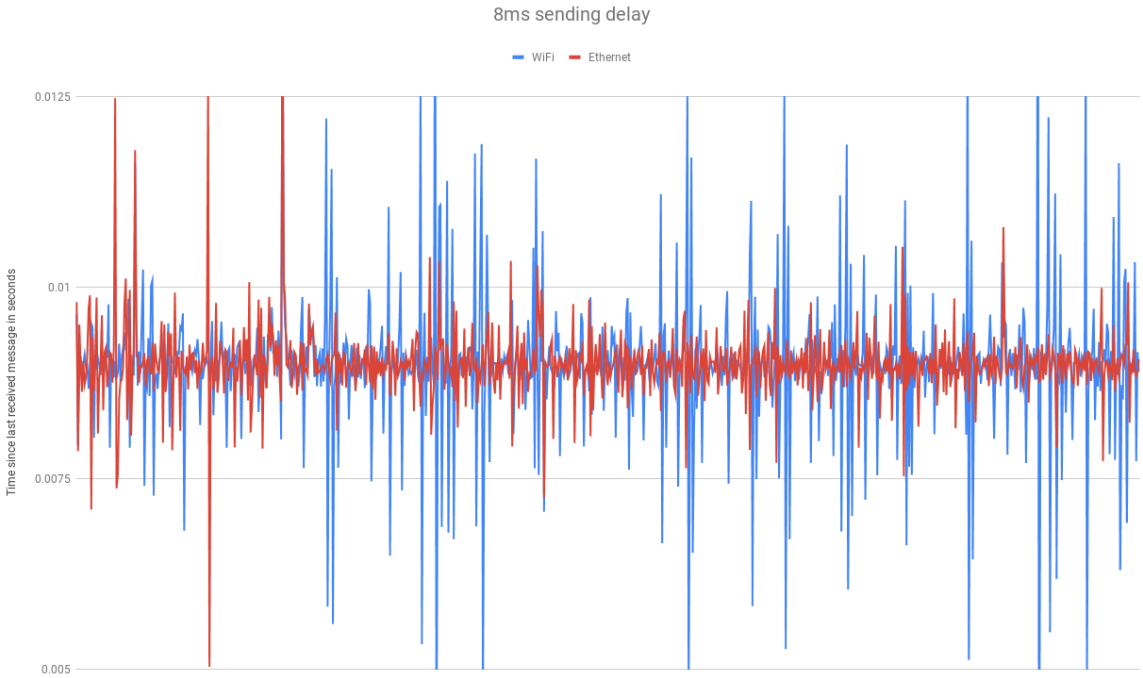


Figure 6.3: 8ms Delay

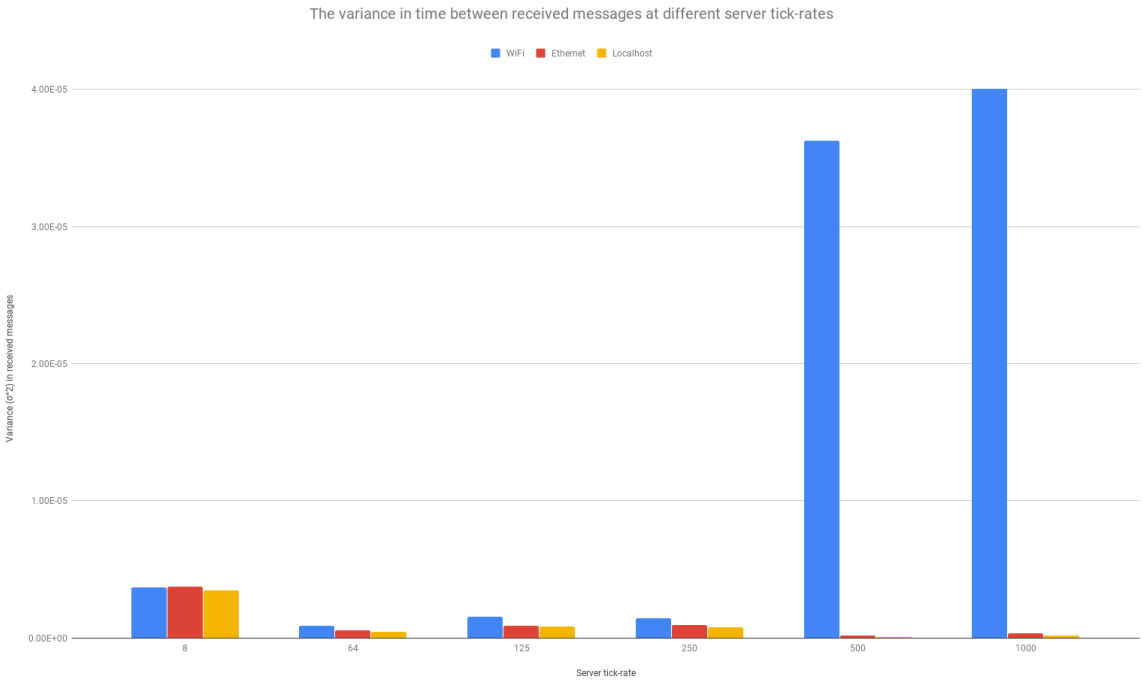


Figure 6.4: Graph showing the variance in the delay between receiving messages.

Chapter 7: Conclusion

This chapter presents a conclusion of the project based on the aims and objectives that were outlined in section 1.3. The following sections will discuss the project's success by determining if each objective has been met and as a result, if the aim of the project has been met.

7.1 Satisfaction of the Aims and Objectives

1. Investigate the common practices and techniques used in modern multiplayer games

   Extensive research into what kind of netcode different examples of popular, modern games use has been done and network development tools in the Unity game engine have been tested through real-world use. The nature of choosing a networking model is a choice that rarely has a concrete single answer. It is often the case that within the same game instance, multiple different networking models are used for synchronising different aspects of the simulation. For example, one of the players might be hosting the physics server for other players, but the rest of the simulation is synchronised through the peer to peer model, taking advantage of faster communication between any two players but having the game physics synchronised between all clients through a single instance.

2. Implement the peer to peer and client hosted networking models as a C++ library.

   The models' logic has been implemented into GNAT just as designed and the interface for using the current implementation is very simple.

   Below is a list of the requirements that have been outlined for the GNAT project:

   - Allow for choice between the peer to peer model or the client hosted model for programs using this code.

     The command line application developed alongside this project uses GNAT and allows the user to choose which networking model to use.
   - The tick rate of data broadcasting should be configurable.

     This has been met as the testing performed in section 6.1, relies on the configurable tick-rate.
   - The only requirement to using the functionality is to clone the library in a known location. Only one import should be needed.

     This has been partially met. The source code successfully compiles into a static library file (GNAT_Core.lib) however the usage of the library, differs slightly. Depending on what networking model is used, either "peer.h", "client.h" or "server.h" has to be included in the project. This only ends up using the necessary classes which should help with compile time on applications that use the library correctly.

- Every class that a user would interact with, should belong to a "GNAT" namespace.

  A user using the library has to correctly use the namespace when referring to a GNAT object. This is important as it provides code clarity if there are several object with a similar name.

3. Investigate issues spawned from computer network communication using the implemented networking models in the GNAT library.

   The investigation into the effect on variance that increasing the server tick-rate has, provided some insightful data.

7.2 What has been established and what could be done further

Throughout the document, the main issues that are encountered when developing real-time, network applications, have been well established and documented. The research into the different methods of mitigating these issues have also been well researched.

Through the investigation of the modern game netcode that has been documented in the section 3.1 and beyond, it can be concluded that the most popular networking model design for games that wish to avoid a dedicated game server, is a peer to peer system with some elements of the game being moved to a client hosted server hosted by one of the peers. The state of this server is cloned periodically but the local copy is not respected and is only useful when a host migration has to occur. If a game is designed in "zones" that allow a player to freely enter and leave, the host migration system can be designed in such a way that as the player approaches another "zone", the migration process begins such that when the player fully leaves, the game does not have to pause to allow for this process to take place.

The implementation of the GNAT library has been done with minimal features and could definitely be expanded further.

7.3 What went well

The investigation into netcode of popular games has proven to provide a lot of insight into the challenges that are encountered and how they can be overcome. Also, the design and implementation of the GNAT library, though not easy, has proved to be an enjoyable challenge and an invaluable learning experience.

7.4 What could have been done better

More experiments could have been performed with more varied scenarios. Some examples include:

- Comparing the bandwidth usage between client hosted and peer to peer models.

- Analysis of performance of the GNAT library and how it could be made more efficient.

7.5 What can be done in the future

There are many potential improvements that could be done to build upon the foundation that have been laid out in this project through the GNAT library. Potential improvements (which have been identified in the section 3.3) include:

- It should be easier to configure the server and port address when using the library. (Ideally, the connection server configuration should be configurable during runtime)
- The library should be easy to set up and forget. (possibly it's own thread could be started that works in the background and keeps the data synchronised).
- It should be possible to send and receive a custom message packet at any stage in the program. (possibly a callback function could be provided that gets called whenever a given message is received)
- Some simulation items should be synchronised automatically. (pointers to struct fields could be given to update values in the struct when a given message is received from a client.)

Appendix A: Networking Model Attributes

A.1 Central Server Model

- ✔ 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 A.1: The attributes of the central server model

A.2 Client Hosted Model

- ✔ 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 negligible 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 acceptable 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 unexpectedly.
- ✘ 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 A.2: The attributes of the client hosted model

A.3 Peer to Peer model

- ✔ 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 inconsistent 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 A.3: The attributes of the peer to peer model

Appendix B: Jitter Variance table

Delay between messages (ms)	WiFi	Ethernet	Localhost
8	3.67E-06	3.74E-06	3.46E-06
64	8.81E-07	5.67E-07	4.61E-07
125	1.54E-06	8.68E-07	8.16E-07
250	1.46E-06	9.19E-07	7.72E-07
500	3.62E-05	1.63E-07	8.40E-08
1000	0.009563789805	3.18E-07	1.84E-07

Table B.1: Table showing the variance values of the jitter seen in test result

Appendix C: Developement Issues

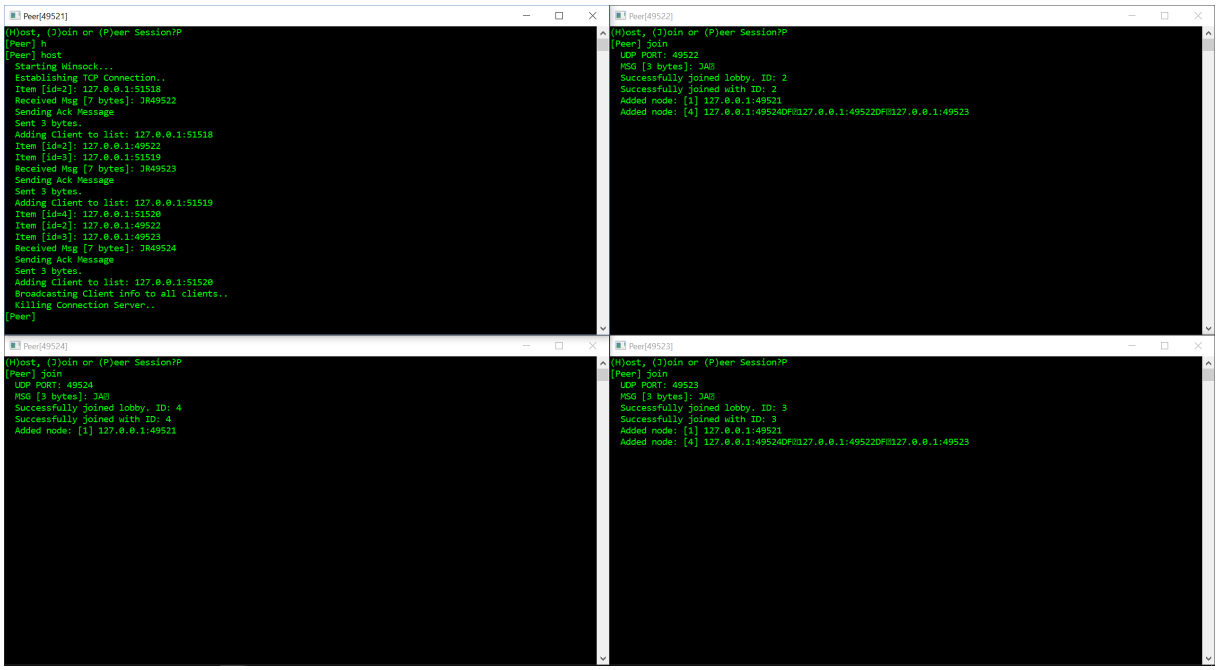


Figure C.1: Messages being broadcast with no delay

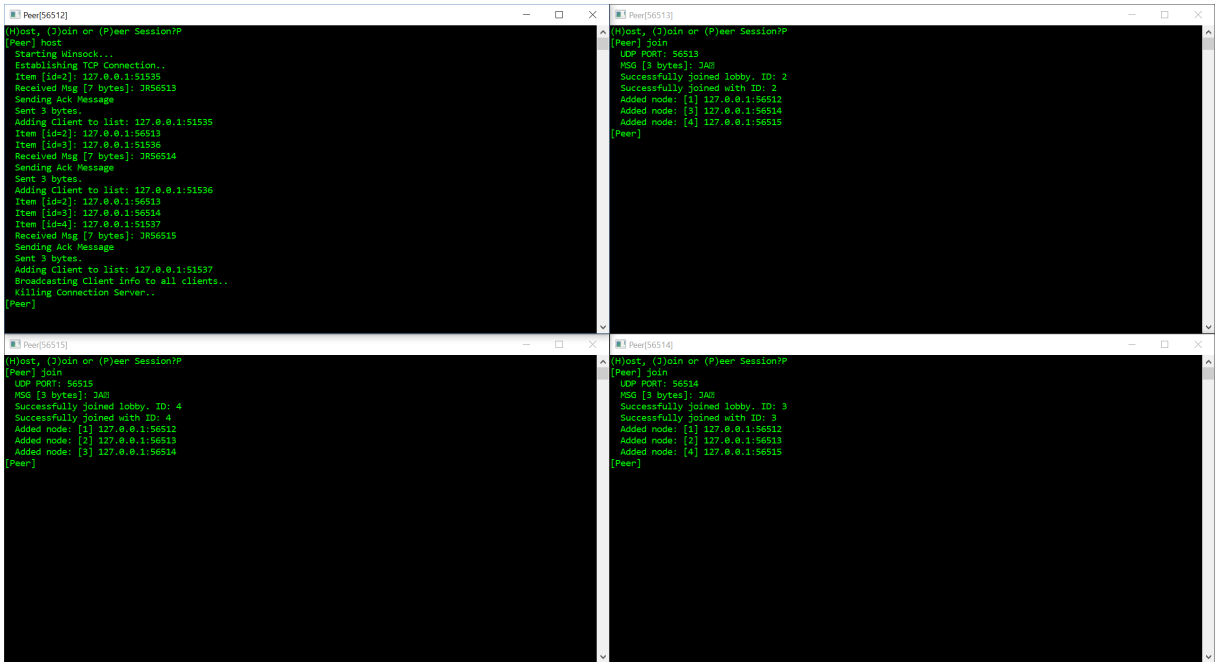


Figure C.2: Messages being broadcast with delay

Appendix D: GNAT Code Snippets

D.1 Pre-compiled Headers

When developing a project that uses code that will not change, such as the std library and winsock, it is inefficient to compile the code for these libraries every time the project code is compiled. To greatly improve compile times, precompiled headers are used.

pch.h

```
// General
#include <string>
#include <iostream>
#include <thread>

// Data Structures
#include <map>
#include <vector>

// WinSock
#include <winsock2.h>
#include <Ws2tcpip.h>
#include <windows.h>

#pragma comment (lib, "ws2_32.lib")

// Custom
#include "log.h"
```

pch.cpp

```
#include "pch.h"
```

D.2 Logging library wrapper

For the logging library, a 3rd party project was used called spdlog. The wrapper is used to use this library in predefined ways.

Log.h

```
#pragma once
#include <memory>
#include <spdlog/spdlog.h>

namespace GNAT {
        class GNAT_Log {
        public:
                static void init();
                static void init_client();
                static void init_server();
```

```
                static void init_peer();
                static void init_connection();

                inline static std::shared_ptr<spdlog::logger>&
                   getConnectionLogger() { return
                   connection_logger; }
                inline static std::shared_ptr<spdlog::logger>&
                   getServerLogger() { return server_logger; }
                inline static std::shared_ptr<spdlog::logger>&
                   getPeerLogger() { return peer_logger; }
                inline static std::shared_ptr<spdlog::logger>&
                   getClientLogger() { return client_logger; }

        private:
                const static int LOG_FILE_SIZE_IN_MB = 5;
                const static int ROTATING_FILE_COUNT = 3;

                static std::shared_ptr<spdlog::logger>
                   connection_logger;
                static std::shared_ptr<spdlog::logger>
                   server_logger;
                static std::shared_ptr<spdlog::logger>
                   peer_logger;
                static std::shared_ptr<spdlog::logger>
                   client_logger;
        };
}

// Loging Macros
#define CONNECT_LOG_FATAL(...)
   GNAT::GNAT_Log::getConnectionLogger()->fatal(__VA_ARGS__);
   std::cout << "  " << __VA_ARGS__ << std::endl
#define CONNECT_LOG_ERROR(...)
   GNAT::GNAT_Log::getConnectionLogger()->error(__VA_ARGS__);
   std::cout << "  " << __VA_ARGS__ << std::endl
#define CONNECT_LOG_WARN(...)
   GNAT::GNAT_Log::getConnectionLogger()->warn(__VA_ARGS__);
   std::cout << "  " << __VA_ARGS__ << std::endl
#define CONNECT_LOG_INFO(...)
   GNAT::GNAT_Log::getConnectionLogger()->info(__VA_ARGS__);
   std::cout << "  " << __VA_ARGS__ << std::endl
#define CONNECT_LOG_TRACE(...)
   GNAT::GNAT_Log::getConnectionLogger()->trace(__VA_ARGS__);
   std::cout << "  " << __VA_ARGS__ << std::endl

#define SERVER_LOG_FATAL(...)
   GNAT::GNAT_Log::getServerLogger()->fatal(__VA_ARGS__);
   std::cout << "  " << __VA_ARGS__ << std::endl
#define SERVER_LOG_ERROR(...)
   GNAT::GNAT_Log::getServerLogger()->error(__VA_ARGS__);
```

```
    std::cout << "  " << __VA_ARGS__ << std::endl
#define SERVER_LOG_WARN(...)
    GNAT::GNAT_Log::getServerLogger()->warn(__VA_ARGS__);
    std::cout << "  " << __VA_ARGS__ << std::endl
#define SERVER_LOG_INFO(...)
    GNAT::GNAT_Log::getServerLogger()->info(__VA_ARGS__);
    std::cout << "  " << __VA_ARGS__ << std::endl
#define SERVER_LOG_TRACE(...)
    GNAT::GNAT_Log::getServerLogger()->trace(__VA_ARGS__);
    std::cout << "  " << __VA_ARGS__ << std::endl

#define PEER_LOG_FATAL(...)
    GNAT::GNAT_Log::getPeerLogger()->fatal(__VA_ARGS__);
     std::cout << "  " << __VA_ARGS__ << std::endl
#define PEER_LOG_ERROR(...)
    GNAT::GNAT_Log::getPeerLogger()->error(__VA_ARGS__);
     std::cout << "  " << __VA_ARGS__ << std::endl
#define PEER_LOG_WARN(...)
    GNAT::GNAT_Log::getPeerLogger()->warn(__VA_ARGS__);
              std::cout << "  " << __VA_ARGS__ << std::endl
#define PEER_LOG_INFO(...)
    GNAT::GNAT_Log::getPeerLogger()->info(__VA_ARGS__);
              std::cout << "  " << __VA_ARGS__ << std::endl
#define PEER_LOG_TRACE(...)
    GNAT::GNAT_Log::getPeerLogger()->trace(__VA_ARGS__);
     std::cout << "  " << __VA_ARGS__ << std::endl

#define CLIENT_LOG_FATAL(...)
    GNAT::GNAT_Log::getClientLogger()->fatal(__VA_ARGS__);
    std::cout << "  " << __VA_ARGS__ << std::endl
#define CLIENT_LOG_ERROR(...)
    GNAT::GNAT_Log::getClientLogger()->error(__VA_ARGS__);
    std::cout << "  " << __VA_ARGS__ << std::endl
#define CLIENT_LOG_WARN(...)
    GNAT::GNAT_Log::getClientLogger()->warn(__VA_ARGS__);
    std::cout << "  " << __VA_ARGS__ << std::endl
#define CLIENT_LOG_INFO(...)
    GNAT::GNAT_Log::getClientLogger()->info(__VA_ARGS__);
    std::cout << "  " << __VA_ARGS__ << std::endl
#define CLIENT_LOG_TRACE(...)
    GNAT::GNAT_Log::getClientLogger()->trace(__VA_ARGS__);
    std::cout << "  " << __VA_ARGS__ << std::endl
```

Log.cpp

```
#include "pch.h"
#include "log.h"
#include <spdlog/sinks/rotating_file_sink.h>
#include <ctime>

namespace GNAT {
```

```
std::shared_ptr<spdlog::logger>
   GNAT_Log::connection_logger;
std::shared_ptr<spdlog::logger>
   GNAT_Log::server_logger;
std::shared_ptr<spdlog::logger> GNAT_Log::peer_logger;
std::shared_ptr<spdlog::logger>
   GNAT_Log::client_logger;

void GNAT_Log::init() {
        spdlog::set_pattern("%^[%T] %n: %v%$");

        try
        {
                peer_logger =
                   spdlog::rotating_logger_mt("CONN",
                   "Logs\\CONN-" +
                   std::to_string(std::time(0)) +
                   ".log", 1024 * 1024 *
                   LOG_FILE_SIZE_IN_MB,
                   ROTATING_FILE_COUNT);
                server_logger =
                   spdlog::rotating_logger_mt("SERV",
                   "Logs\\SERV-" +
                   std::to_string(std::time(0)) +
                   ".log", 1024 * 1024 *
                   LOG_FILE_SIZE_IN_MB,
                   ROTATING_FILE_COUNT);
                peer_logger =
                   spdlog::rotating_logger_mt("PEER",
                   "Logs\\PEER-" +
                   std::to_string(std::time(0)) +
                   ".log", 1024 * 1024 *
                   LOG_FILE_SIZE_IN_MB,
                   ROTATING_FILE_COUNT);
                client_logger =
                   spdlog::rotating_logger_mt("CLNT",
                   "Logs\\CLNT-" +
                   std::to_string(std::time(0)) +
                   ".log", 1024 * 1024 *
                   LOG_FILE_SIZE_IN_MB,
                   ROTATING_FILE_COUNT);
        }
        catch (const spdlog::spdlog_ex& ex)
        {
                std::cout << "Log initialization
                   failed: " << ex.what() << std::endl;
        }
}

void GNAT_Log::init_client() {
```

```
        spdlog::set_pattern("%^[%T] %n: %v%$");

        try
        {
                client_logger =
                   spdlog::rotating_logger_mt("CLNT",
                   "Logs\\CLNT-" +
                   std::to_string(std::time(0)) +
                   ".log", 1024 * 1024 *
                   LOG_FILE_SIZE_IN_MB,
                   ROTATING_FILE_COUNT);
        }
        catch (const spdlog::spdlog_ex& ex)
        {
                std::cout << "Log initialization
                   failed: " << ex.what() << std::endl;
        }
}

void GNAT_Log::init_server() {
        spdlog::set_pattern("%^[%T] %n: %v%$");

        try
        {
                server_logger =
                   spdlog::rotating_logger_mt("SERV",
                   "Logs\\SERV-" +
                   std::to_string(std::time(0)) +
                   ".log", 1024 * 1024 *
                   LOG_FILE_SIZE_IN_MB,
                   ROTATING_FILE_COUNT);
        }
        catch (const spdlog::spdlog_ex& ex)
        {
                std::cout << "Log initialization
                   failed: " << ex.what() << std::endl;
        }
}

void GNAT_Log::init_peer() {
        spdlog::set_pattern("%^[%T] %n: %v%$");

        try
        {
                peer_logger =
                   spdlog::rotating_logger_mt("PEER",
                   "Logs\\PEER-" +
                   std::to_string(std::time(0)) +
                   ".log", 1024 * 1024 *
                   LOG_FILE_SIZE_IN_MB,
```

```
                        ROTATING_FILE_COUNT);
                }
                catch (const spdlog::spdlog_ex& ex)
                {
                        std::cout << "Log initialization
                           failed: " << ex.what() << std::endl;
                }
        }

        void GNAT_Log::init_connection() {
                spdlog::set_pattern("%^[%T] %n: %v%$");

                try
                {
                        connection_logger =
                           spdlog::rotating_logger_mt("CONN",
                           "Logs\\CONN-" +
                           std::to_string(std::time(0)) +
                           ".log", 1024 * 1024 *
                           LOG_FILE_SIZE_IN_MB,
                           ROTATING_FILE_COUNT);
                }
                catch (const spdlog::spdlog_ex& ex)
                {
                        std::cout << "Log initialization
                           failed: " << ex.what() << std::endl;
                }
        }
}
```

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.

Carter, Chris et al. (2009). "Networking Middleware and Online-Deployment Mechanisms for Java-Based Games". In: *Transactions on Edutainment II*. Ed. by Zhigeng Pan et al. Berlin, Heidelberg: Springer Berlin Heidelberg. ISBN: 978-3-642-03270-7. DOI: 10.1007/978-3-642-03270-7_2. URL: https://doi.org/10.1007/978-3-642-03270-7_2.

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

Jefferson, David R. (July 1985). "Virtual Time". In: *ACM Trans. Program. Lang. Syst.* 7.3, pp. 404–425. ISSN: 0164-0925. DOI: 10.1145/3916.3988. URL: http://doi.acm.org/10.1145/3916.3988.

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.

Xie, Jingming (2012). "Research on key technologies base Unity3D game engine". In: *2012 7th international conference on computer science & education (ICCSE)*. IEEE, pp. 695–699.

Zenke, M. (2008). "Land of fire: The rise of the tiny MMO". In: