Reusable Networking Library for Game Development

Final report

Submitted for the BSc in

Computer Science for Games Development

April 2023

by

Henry Paul

Word count: 15,680

1 Abstract

This project covers the investigation of current practices for networked game development and demonstrates those practices through the development of a C# networking library primarily targeting the MonoGame framework.

The result of this is a networking library that can be freely used or expanded upon by any game developer who wishes to do so. In terms of using the library, it has been designed to require as little effort, previous knowledge and experience of networking from the developer as possible.

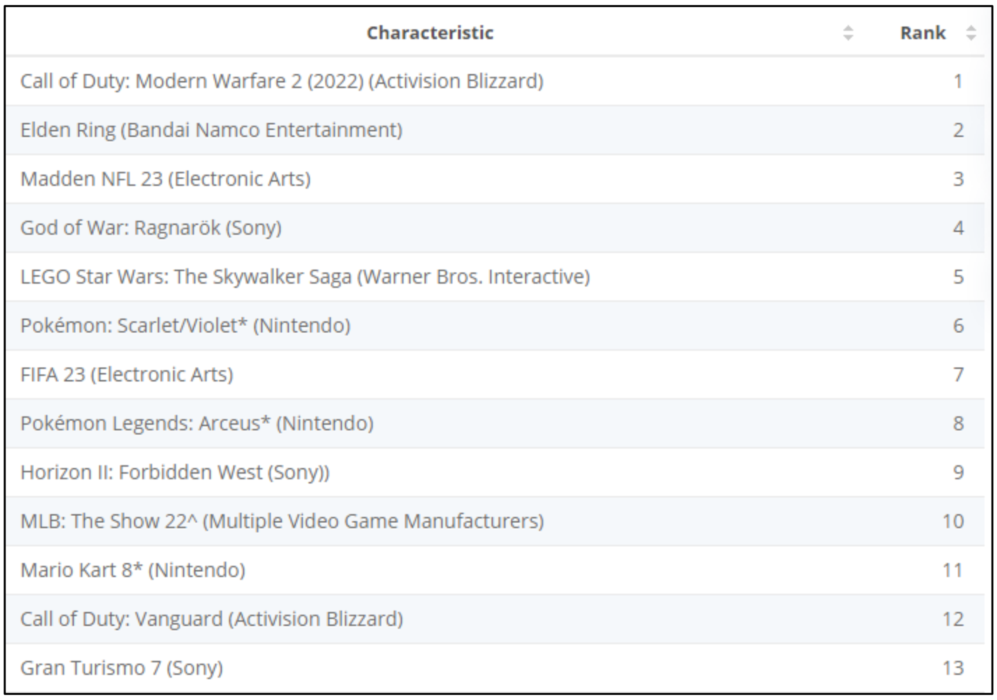
Furthermore, this document will critically evaluate the choices made during development and highlight any areas for potential improvements. One of the ways it will be evaluated is by measuring the performance of the implementation, noting differences in metrics such as frames per second, whilst there is a networked session active in comparison to a purely local session.

Contents

1 Introduction

1.1 Background to the project

According to data retrieved from statista.com, 9 of the 13 best-selling games in the US in 2022 offered some form of real-time multiplayer gameplay. Specifically, games ranked number 1, 2, 3, 6, 7, 10, 11, 12 and 13 in figure 1.1.



Characteristic	Rank
Call of Duty: Modern Warfare 2 (2022) (Activision Blizzard)	1
Elden Ring (Bandai Namco Entertainment)	2
Madden NFL 23 (Electronic Arts)	3
God of War: Ragnarök (Sony)	4
LEGO Star Wars: The Skywalker Saga (Warner Bros. Interactive)	5
Pokémon: Scarlet/Violet* (Nintendo)	6
FIFA 23 (Electronic Arts)	7
Pokémon Legends: Arceus* (Nintendo)	8
Horizon II: Forbidden West (Sony))	9
MLB: The Show 22^ (Multiple Video Game Manufacturers)	10
Mario Kart 8* (Nintendo)	11
Call of Duty: Vanguard (Activision Blizzard)	12
Gran Turismo 7 (Sony)	13

Figure 1.1: Best-selling video games in the United States in 2022, by dollar sales (J. Clement, 2023)

Online gaming is undoubtedly a largely popular and rapidly growing industry which is why so many development studios are opting to switch their focus to multiplayer even if they have a long history of creating singleplayer games. One controversial example being Bethesda, who released Fallout 3 and 4. Two singleplayer games with a heavy focus on story, with the latter releasing in 2015. However, in 2018, they released Fallout 76, an online focused game within the same vein as the previous entries. The game released to a very poor reception due to a variety of reasons, but one of the most egregious being the broken nature of the game, likely brought on by the addition of online multiplayer.

Todd Howard, the director of Bethesda Game Studios, even acknowledged the criticism saying the following in an interview with IGN “That was a very difficult development on that game to get it where it was,” “We were ready for… a lot of those difficulties that ended up on the screen. We knew, hey look, this is not the type of game that people are used to from us and we’re going to get some criticism on it. A lot of that is very well-deserved criticism.” (J. Knoop, 2019).

The focus of this project is to explore the technology behind online gaming and develop a networking library that is versatile enough to be used in various projects, with the only limitation being the framework used. It was originally designed to target MonoGame, a C# based game framework. It could be used in a more fleshed out game engine such as Unity since that also uses C#. However, the differences in how Unity is designed may require some changes, albeit small, to the library to work properly. The goal is that a game developer without any previous knowledge or experience with networking should be able to make use of this library with minimal effort.

To demonstrate the library, a small prototype game has been created that allows multiple players to choose their own player sprite and join a peer-to-peer session with other players.

1.2 Aims and objectives

1.2.1 Objective 1: "Develop a groundwork for the game using C# and MonoGame"

The first objective of this project was to create a basic project in MonoGame that would later be expanded into the demonstration game and was also used to experiment with ideas during the initial design process. It was a simple 2D scene in which the player took control of a circle sprite texture and moved it around the screen using the arrow keys.

1.2.2 Objective 2: "Create a reusable networking component as a DLL file that is capable of providing multiplayer features to an existing game"

The second objective was to develop the networking library. The goal was to make sure the library was useable in multiple different MonoGame projects without needing any game specific code adjustments being made to it. As a DLL file, the game developer is able to add the file to their project as a dependency, giving them access to its methods and data. At a minimum, it was supposed to provide the ability to create and manage peer-to-peer game sessions. However, it was desired to include other networking types and features such as support for game sessions running on dedicated servers.

1.2.3 Objective 3: "Create a prototype game to test the networking component"

This objective is essentially an extension of the first objective, turning the initial MonoGame project into more of a game whilst also making use of the networking library to demonstrate its features. It should allow multiple clients to connect to each other and have their local actions replicated across all connected clients. Each player would take control of a sprite and be able to move around the scene. This game should demonstrate all the features of the networking library.

1.2.4 Objective 4: "Testing and evaluation of performance"

Originally, this objective detailed the use of a test plan and manual testing. However, during development, a switch was made to unit testing, where each feature of the library would be tested independently by code. Additionally, the performance of the library should be tested and evaluated by measuring change in FPS as more clients join a session.

1.2.5 Secondary objectives

If time allowed, these objectives were planned to be worked on:

- Add more networking features to the library such as anti-cheat, leaderboards or matchmaking
- Expand DLL to support other engines such as Unity
- Create a full game experience to demonstrate a real use case example of the networking library

2 Literature review

2.1 Introduction

This research will detail the current knowledge of online gaming and how multiplayer features are implemented in games.

2.2 Networking Architectures

2.2.1 Peer-to-peer

Peer-to-peer (P2P) is a networking architecture that allows the transfer of information between devices, with each device having equal permissions and responsibilities for processing data (Webroot, 2004). In the early days of the Internet, it was commonly used in file sharing services such as Napster and LimeWire. However, this technology can also be applied to gaming.

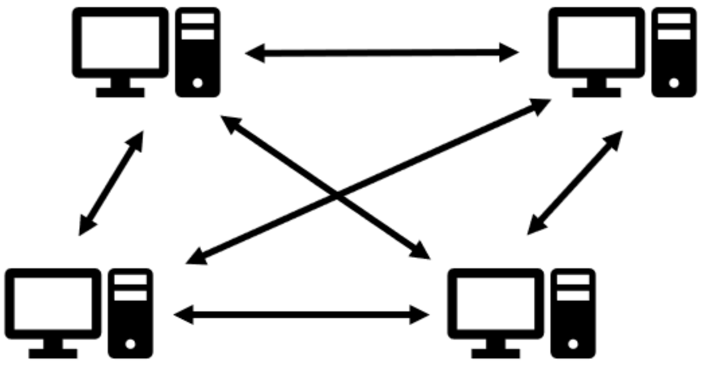


Figure 2.1: Peer-to-peer network

Peer-to-peer gaming works by establishing multiple connections between devices in the mesh pattern shown in figure 2.1. They will all be responsible for sending and receiving data to ensure synchronisation of the game, with no central server acting as authority (Noland, 2019). For example, in a network of 4 connected devices (with each device representing a player), each player will run the game and any calculations locally and tell the other players what they are doing. So, if player 1 decided to move forward in the game, it would send a packet containing this information to the rest of the players in the network and then process that player input locally as demonstrated in figure 2.2. Upon receiving this packet, the other players would then process player 1's input in their local game accordingly.

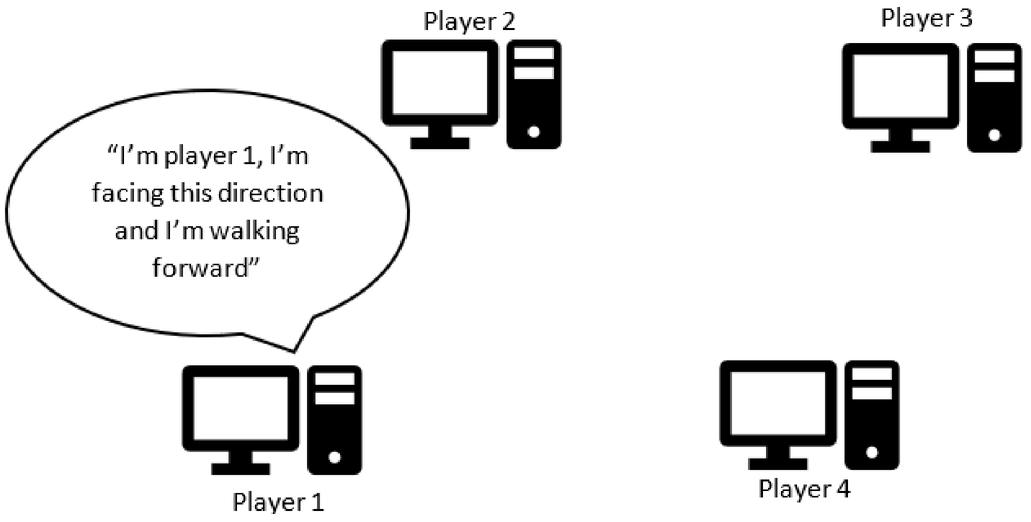

Player 2
Player 3
"I'm player 1, I'm facing this direction and I'm walking forward"
Player 1
Player 4

Figure 2.2: Peer-to-peer communication

Players 2 – 4 receive the message that player 1 sent and update their own running version of the game with that information. These players would then do the same, sending any updates they make to the rest of the devices in the network.

This type of networking has its advantages, for example, it is cheap. The developer doesn't need to spend any money running and maintaining dedicated servers because it is all handled by clients establishing their own connections between each other. This also means that the game and its multiplayer features can technically run forever, as long as there are players and a functioning Internet, as it would not rely on any external servers that could be shut down by the developer.

One example of this happening is the game "Evolve". Evolve was a multiplayer game in which 4 players fought co-operatively against another player who would be controlling a monster, with this player's objective being to kill the others. The game never managed to find the success needed to be maintained and it was decided that the servers would go down on the 3rd September 2018, over 3 years after its initial release in February 2015. However, the game wasn't unplayable after the servers shut down. Instead, the game would now rely on peer-to-peer matchmaking (Treese, 2018).

However, there are many disadvantages to this type of networking. So much so, that larger developers will lean toward using dedicated servers as an alternative if they can afford it. For example, security can be a big issue with this architecture. This is because each of the devices will essentially have a list of IP addresses associated with the devices it is connected to, giving users malicious capabilities, and revealing your location.

One of the biggest issues with this implementation is preventing cheating. There are many ways a player could manipulate their own local game in ways that affect the overall game for other players in the session. For example, this could simply be by gaining access to information the cheater would otherwise not be able to view. Take for instance an online poker game. The general idea of poker is that each player is given a hand of cards and players can only see their own cards. Based on the perceived strength of their own hand of cards, they would place bets reflecting their own confidence that their hand is stronger than anyone else's. Now, if a player could see everyone's cards, all of a sudden it would be easier for them to make successful bets. This is one problem of peer-to-peer gaming, since the data is all stored locally, it opens up the possibility of players finding ways to observe this data when they otherwise wouldn't be able to (Neumann et al., 2007).

Furthermore, as each device is responsible for sending and receiving data, a player can manipulate this data flow. Let's say player 1 is a cheater in a P2P FPS game. Player 2 is claiming they have fired a bullet directly at player 1, it should be a guaranteed hit. However, player 1 can essentially choose to ignore this and decide not to process any of these hits. On player 2's screen, they are firing bullet after bullet at player 1, each giving them the appropriate on-screen feedback that they are hitting them. But player 1 has decided that the bullets aren't hitting them and thus they are telling the other player that they are taking no damage.

2.2.2 P2P with authority

P2P with authority is similar to the previously mentioned peer-to-peer architecture in that it shares the same advantage of not needing to maintain a dedicated server. Where peer-to-peer has no centralised host/server, P2P with authority assigns one of the players in the network as the "host" (Roxl, 2021). This can also be referred to as client-server, where the server is also acting as a client. The player with the strongest connection should be picked as the host because, if the host has a slower connection, everyone else will hindered by the low speeds. Like the previously detailed peer-to-peer method, each player will communicate when they are giving an input. This time though, they will only communicate with the host instead of each player communicating with every player on the network. Furthermore, the only person running the game code is the host.

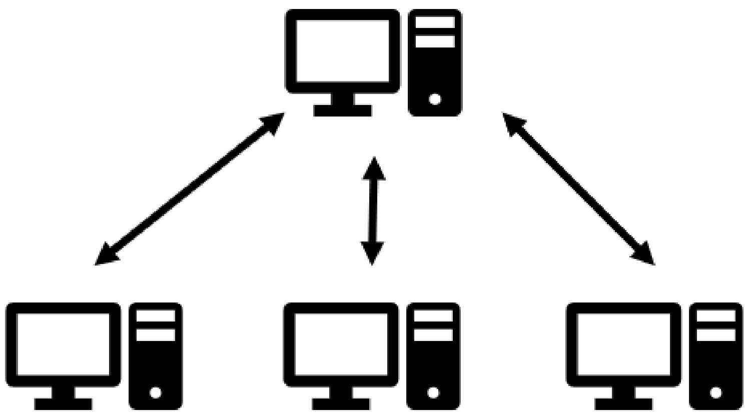


Figure 2.3: P2P with authority

So, let's say player 1 is the host and player 2 is moving forward. Player 2's device would detect the user input and tell player 1 (the host) that player 2 is moving forward. This input would then be processed on the host device, with player 2's position updated on the host client. The host would then send all the information of the game needed to render the current scene, to each of the players in the network. Essentially, everyone, except the host, simply acts as a terminal to receive information about the current state of the game, and render that information, without running any game logic locally. This means that the host acts as an authoritative figure in the network, the connected clients can't "disagree" on the current state of the game and is all decided by the host.

However, if the host has a bad connection, then the rest of the players will not be able to be updated on the current state of the game quick enough. In addition to this, the host will always have an unfair advantage over everyone else. This is because they will have zero latency (delay) and everyone else will have varying ranges of latency, mainly depending on their geographical distance to the host. This is why both peer-to-peer and P2P with authority can't be recommended for competitive games if it can be avoided.

Another disadvantage is what happens when the host disconnects. All of a sudden, game logic is no longer being run by anyone, as this would have been done on the hosting client. To get around this, something called "host migration" will begin. This is the process of transferring the hosting responsibilities to another device in the network (Microsoft, 2021). Because of this, gameplay would have to be suspended whilst this transfer takes place. This can be frustrating for players as it hinders their ability to have a seamless gameplay experience.

2.2.3 Dedicated server / client-server

A dedicated server model, also a variation of client-server, works by having a dedicated device (the server) run the game code and handle connections to clients. This is very similar to the P2P with authority approach, except, instead of appointing one of the clients to be the host, there is no player host. This means none of the players have authority over the others, and no one should have an unfair advantage because they were chosen to be the host of the session. A dedicated server will take the role of the game host (Velimirovic, 2021), running the game code and processing user inputs, before sending the results back to the clients so that the game can be updated and rendered for each player.

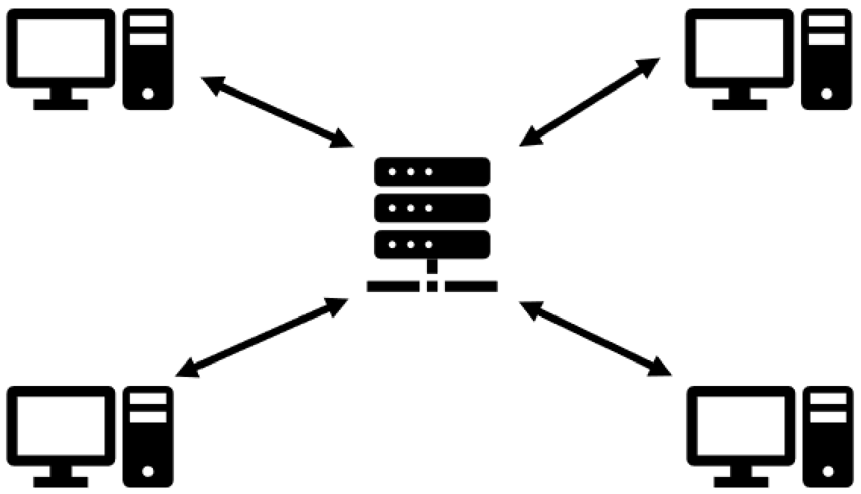


Figure 2.4: Client / server model

Usually, this would include things like physics simulations. If each client was responsible for calculating these things independently, it is likely that each player would get different results, even just slightly. This is because of floating point errors. Floating point calculations aren't fully accurate, Zou et al. (2015) say this is because floating point numbers use a finite number of digits to represent a real number. This results in the possibility of each client having differing values for the same thing. For example, let's say that a player throws a grenade towards another player. When calculating the trajectory for that grenade and where it should land, multiple variables would be taken into account. Things like gravity, throwing force, air resistance and ground friction could all change how that grenade moves through the environment. If the two players had different values for each of these variables, even if the difference is small, each player would have different ideas of where that grenade should end up. On a dedicated server, all of those calculations are done by one machine, and that machine has authority over everything else. Meaning, every player should see the same results. Provided they have a good enough ping. Each player's ping is dependent solely on themselves and their geographical distance to the server. Because of this, it is important to have multiple servers around the world for different regions, so that any player from any region can enjoy a playable experience. With this in mind, it is clear that this can very quickly become very expensive to run and maintain. Therefore, the use of dedicated servers is not recommended for a team on a small budget. However, it could still be viable if the expected number of players is small and locked to a specific region. But for games with a large number of players and regions, dedicated servers are only recommended for teams with a large budget.

2.3 Networking protocols – TCP vs UDP

2.3.1 What is TCP?

TCP "transmission control protocol" is a networking protocol used widely across the internet. It is used in HTTP (Hypertext Transfer Protocol: used to load web pages), SMTP (Simple Mail Transfer Protocol: email systems) and FTP (File Transfer Protocol: transferring files between a server and client) (Fortinet, 2023). It is a common choice for these applications due to its reliability and relative ease of use. This is because TCP can be implemented like reading and writing to a file. It functions as a stream, so when sending data, you would write to the stream as if it was a file, TCP would then break that data down into packets and send them across the network (Fiedler, 2008b). On arrival, the packets are reconstructed into its original form so that it can be read, like a file, by the recipient. Furthermore, all packets are guaranteed to arrive at their destination and will arrive in order. So far, it sounds like TCP is a good choice, but this reliability and ease of use comes at a cost when used in time sensitive situations, which will be detailed later in this document.

2.3.2 What is UDP?

UDP "user datagram protocol" is a more barebones protocol in comparison to TCP. It is a very thin layer over IP and doesn't add a lot of features (Fiedler, 2008b). UDP does not have built in support for connections, this has to be coded manually if it is a desired feature. It also has no guarantee of ordering, or even a guarantee that the packets will even arrive. Additionally, it is a lot harder to implement. Unlike TCP, packets have to be created manually. After a packet has been created, you must specify an IP address and a port number to send that specific packet to. This packet is then sent across the network, transferred between computers until it arrives at the destination or is lost. The recipient of the packet will listen on the specified port number and wait until they receive the packet. However, they will also be listening for any packets that arrive on that port number due to the fact that there is no direction connection between the sender and receiver. UDP is unreliable, just like the IP layer it is built upon. This means that if you send a packet, there is no guarantee it will arrive at the destination. To make things even harder, UDP doesn't have a built-in acknowledgement system like TCP does. Therefore, you won't even know if the packet has arrived or been lost, unless this feature is coded manually.

2.3.3 Which is better?

So far, it sounds like TCP is the obvious choice as it provides more features and is easier to use. However, before making this choice, it is important to understand how all of these advantages of TCP can actually be a bad thing in this use case.

2.3.3.1 How does TCP actually work?

Both UDP and TCP are built upon IP. Unlike UDP, TCP hides all of the complexities of IP and provides the user with some useful features. As previously mentioned, TCP is a stream protocol, meaning bytes are written to and read from a stream on the application side, whilst TCP handles everything else. Since IP is built on the use of packets, and TCP is built on IP, it must break the stream of data into packets that can be sent across the network. This is all done automatically, meaning the developer using the protocol doesn't have to worry about this. Internally, it will queue up the data in the stream. When there is enough data in the queue, it will send the packet to its destination (Fiedler, 2008b). Immediately, this is a clear problem.

In any situation that could require the sending of small packets, TCP may decide that the packet isn't big enough to be sent, instead waiting for the packet to be big enough. For games specifically, this would be a major problem as the main goal is to receive the data as quickly as possible. If this time is spent waiting for the data to be a big enough size, game updates would reach players at infrequent rates. However, this problem is easily fixed with the use of a built-in property called "TCP_NODELAY". This ensures that packets are always sent immediately instead of waiting for a specific size.

This still doesn't make TCP the desirable choice though. Even with this improvement, it is still too slow for handling time sensitive data due to the way it detects lost packets and the way it ensures packets arrive in the correct order. How does this work?

When a packet is sent using TCP, the sender will wait until it receives an acknowledgement (another packet sent from the recipient, to confirm that they have received the packet sent by the original sender) before sending the next one (Fiedler, 2008b). If an acknowledgement is not received after some time, that packet will be declared lost. Once this happens, that packet will be sent again, and the process repeated until the acknowledgement is received. Only once this happens, will the next packet be sent. Clearly, this is problematic as the whole protocol will come to a stop and wait for a single packet to successfully reach its destination. If any newer data is written to the stream, it will be put into a queue until the lost packet has been resent.

This is why TCP should never be used for time critical applications such as games. In a networked game, user inputs will be sent to a server (whether it be player hosted or a dedicated server) every frame. The server will then process these inputs, update the game state, and send the results back to the players, every frame. If TCP is used for this transfer of data, and a packet is lost, all players would temporarily stop receiving game updates, making things appear to be motionless and the server would stop receiving player inputs, causing the game to feel unresponsive on the client side.

UDP doesn't have these issues, making it the better choice for this application. However, it also doesn't share the same functionality that would still be desirable which means that some work will need to be done to add these things manually.

2.3.4 Improving upon UDP

Since UDP doesn't have a lot of the benefits included by TCP, the protocol will need to be improved upon for this use case.

2.3.4.1 Virtual connection

Since UDP is connectionless, a socket can send and receive packets from multiple devices. In this case, this is bad. If a client is listening to all packets that are received, it will try to process each one of them as if it belonged to the game protocol, even if it didn't. This is why a virtual connection needs to be created, so that only the game specific packets will be processed. This example, proposed by Fiedler (2008b), will demonstrate how a virtual connection can be made between two computers.

First, consider how to filter packets that belong to the game protocol. Ideally, this can be done by filtering based on the IP that sent a packet. When a packet is received, it will include the IP address it was sent from. So, if that IP matches the address of the other computer, process it. If not, discard the packet. This works great for the client since the client will know the IP address of the server in advance. However, the server will not know the IP address of a connecting client in advance. To fix this, a protocol ID should be used. By prefixing the packet with a unique number (unique to the game protocol), the application can read the first four bytes of any incoming packet and determine whether or not it should ignore it by comparing them against the protocol ID. If it's a match, ignore the first four bytes and process the rest of the packet. If it is not a match, ignore the whole packet, as it does not belong to the game protocol.

To establish a connection then, the server can take the first packet it receives with the correct protocol ID and store the IP address it was sent from. From that point on, the server can now filter packets by IP address instead.

It is also important to detect when a disconnection has occurred. This can be done simply by doing the inverse of establishing the connection, where a connection is defined as receiving packets, a disconnection can be defined as not receiving packets. By keeping track of the time since receiving a packet, a disconnect can detected by checking if this time has exceeded a specific value, if it has then the devices can no longer be considered connected. If the client disconnected, the server should remove it from its list of connections. If the server disconnected, the client should stop sending packets to the server.

To allow multiple connections, the server should filter packets by both the IP address and the protocol ID. The protocol ID should only be used when initialising a connection and IP address used when a connection already exists. So, for multiple connections, the server should have a list of currently connected clients represented by their IP address. When receiving a packet, current connections should have priority over new connections by first comparing the address of the packet against the list of clients. If the address belongs to a client in the list, process the packet as normal. However, if there is no match in address, the server should then attempt to filter by protocol ID. If the ID is a match, then add the address of that packet to the list of connections. If not, ignore the packet, as it does not belong to the game.

2.3.4.2 Reliability

TCP has a built-in acknowledgement and ordering system (Khan Academy, 2020) which makes it very reliable in comparison to UDP which is missing this functionality. To make UDP a more reliable protocol (whilst still being appropriate for time critical data), a custom implementation of acknowledgements needs to be created.

2.3.4.2.1 Sequence numbers

First, an understanding of how TCP accomplishes this is required. This is done by using something called sequence numbers. This is an integer value that is included in every packet, representing how recent the packet is. It starts at 0 for the first packet sent, incrementing by 1 with every packet. Because UDP doesn't guarantee ordering, the 100th packet sent is not necessarily the 100th packet that will be received. Sequence numbers can fix this issue by allowing the application to order the packets based on their sequence number.

2.3.4.2.2 Acknowledgements

An acknowledgement (or "ack") is a method of detecting when a sent packet has successfully arrived at its destination (Awati, 2023). To begin, a simple implementation of this will be detailed. Whilst this won't be a suitable way to accomplish this, it is important to understand the concept at a base level before considering a better version which will be detailed later. Both solutions were proposed by Fiedler (2008a).

2.3.4.2.2.1 Simple acks

Simply put, this works by taking the previously mentioned sequence number of a packet and sending that number back to where it came from.

Each computer should store two integer numbers for this. A local sequence number and a remote sequence number. Every time a computer sends a packet, they should also increment their own local sequence number by 1. This keeps track of how many packets that computer has sent. The remote sequence number represents the most recent packet they have received, which is done by comparing a newly received packet's sequence number against the current remote sequence number. If the packet is more recent, then the remote sequence number should be updated to the packet's sequence number.

When a packet is constructed, the local sequence number of the sending computer becomes the sequence number of the packet and the remote sequence number held on that computer becomes what is called the "ack" of the packet which can be processed by the receiving computer, telling them that the packet associated with that ack arrived successfully. However, the problem with this system is that it will only ever work if one packet is sent for every packet received due to the fact that only one ack number can be held in a packet. This can be a problem when both computers are sending packets to each other at different rates. For example, if a client is sending 30 packets/second to a server, and that server is sending 10 packets/second back, the server will need to make space for 3 acknowledgements in each packet to keep up.

Furthermore, if a packet containing an acknowledgement gets lost, the original sender of the packet that was supposed to be acknowledged would think their packet never arrived, when instead it did arrive, but the acknowledgement packet got lost.

2.3.4.2.2.2 Reliable acks

This solution is a much bigger divergence from TCP but is also much more suitable than the previous solution. Unlike TCP, this solution should never stop and resend a lost packet. Packet "n" is sent once, followed by "n+1, n+2…" and so on. Instead, if a packet is lost, the application should determine whether or not a new packet containing the lost data should be created and sent. TCP would stop here and resend the packet, instead, the flow of packets should continue and choose to deal with any losses later, where there would be a gap in a list of acknowledged packets.

The problem with simple acks is that there isn't enough space in a packet to acknowledge enough packets and the case of an acknowledgement being lost. To fix this, more acks need to fit in the packet which can be done through the use of a bitfield. Each packet sent contains an ack and an ack bitfield. The single ack number works as it did before, representing the current remote sequence number (most recently received packet). However, the ack bitfield is 32 bits, with each bit representing an acknowledgement. The first bit in the field specifically represents ack-1, so if ack was 100 (most recently received packet was 100) and the first bit is set (1 or true) then that would mean packet 99 is being acknowledged in this packet. This continues for another 31 bits, meaning 33 acknowledgements can be made in a single packet as opposed to 1. If the acknowledgements were to be held in the packet similar to the simple ack (just an integer value for each sequence) then this would take up 132 bytes. As well as being able to fit more acknowledgements, this solution allows each acknowledgement to be sent 33 times. This means that 100% of those 33 packets would have to be lost, for an acknowledgement to be lost, instead of just the single packet being lost with the previous method.

Step by step breakdown of this solution:

- Send a packet, increment local sequence number
- When packet is received, compare sequence number of packet against the current remote sequence number (most recently received packet). If packet is more recent, update remote sequence number
- Compose a packet header
    - Current local sequence number becomes sequence number of packet
    - Current remote sequence number becomes ack of packet
    - Ack bitfield is set by looking at a queue of the 33 most recently received sequence numbers. Set bit n to 1 if remote sequence number – n is in the queue. If not in queue, set to 0
- When a packet is received, scan ack bitfield and determine which packets have been lost

2.3.4.3 Packet loss detection

With this reliable acknowledgement system, it is now possible to detect when a packet is lost. This is done in a similar way to connection timeouts, if no acknowledgments have arrived for a specific packet in a certain amount of time, then the packet can be considered lost. Whilst there is 100% certainty of packet arrival (receiving an acknowledgement confirms without a doubt that a packet arrived), packet loss can be detected with a reasonable amount of certainty. Because of this relative uncertainty, it is important to make use of some sort of message ID in packets so that duplicate data can be discarded. If a packet is detected as lost when it actually isn't, another packet could be sent with the same data. The receiving computer would then receive two packets containing the same data but would assume they are not duplicates and process them both as a new packet anyway.

2.3.4.4 Sequence number wrap-around

Sequence number wrap-around is a problem that only becomes an actual issue when using 16-bit integers for acknowledgements and sequence numbers. When using 32-bit integers for this, 30 packets would have to be sent every second, for over four years straight before this becomes an issue. However, using 16-bit integers can save on bandwidth, but this causes sequence number wrap-around to become an issue in just half an hour with that same send rate (Fiedler, 2008a).

This problem is caused by the value of a sequence number becoming too large to be stored as an integer. In this case, the number needs to be reset to 0, but now the sequence of packets won't be ordered correctly. The new "0" packet is more recent than packet "65,353", but the application doesn't know that and will continue to look for packets that have a higher numerical sequence number than 65,353.

To solve this issue, some kind of algorithm needs to be implemented to detect when the sequence has wrapped. Let's take a one-byte integer example, in which each acknowledgement and sequence numbers is stored in a one-byte integer. The maximum value of this integer would be 255, and this is what the sequence would look like when wrapped:

"…252, 253, 254, 255, 0, 1, 2, 3…"

To implement a function that determines when this has happened, first take two sequence numbers, the current remote sequence number (most recent packet arrival) and the next sequence number that is currently being processed. If the difference between the numbers is less than half the maximum sequence number, in this case 255, then the numbers must be close together in the sequence. If this happens, then just compare the numbers as usual, with the bigger number being the most recent packet. If the numbers are far apart on the sequence, the difference between them will be more than half the maximum value. In this case, the opposite needs to be done, with the smaller number now being considered more recent than the bigger number.

2.3.4.5 Congestion avoidance

When packets are sent without flow control, there is a risk of flooding a connection which can cause severe latency problems due to how any routers between the sending and receiving device are buffering packets. Routers will attempt to deliver all packets and prioritises that over dropping them. When a router is overloaded with packets, they will be put into a queue which will slow everything down (Fiedler, 2008a). Since this is a problem induced by the design of routers, there is no way to solve this directly, but the problem can be avoided with the use of flow control. First, it is important to understand how to measure round trip time (RTT).

2.3.4.5.1 Measuring round trip time (RTT)

This is a measurement of how long it takes for a packet to be sent and an acknowledgement of its arrival to be received by the original sender. To calculate this, a queue can be created to track all recently sent packets. Every time a packet is sent, add it to a local queue containing the sequence number and the time it was sent. Every time an ack is received, find the associated packet in the queue, and compare the difference in time between receiving the ack and sending the packet. The result will be the RTT for that packet, to gain an understand of the overall current round-trip time, this value needs to be smoothed out since the value will vary due to network jitter. This can be done with a smoothed moving average.

Once a specific maximum round trip time has passed, a packet should be removed from the queue to avoid a constant growth of the queue, for example, 1 second maximum RTT. The basic idea of congestion avoidance is if RTT is too large, send less packets. If RTT is small, send more packets.

2.3.4.5.2 Binary congestion avoidance system

This system, proposed by Fiedler (2008a), has two modes: "good" and "bad". Assuming that packets are 256 bytes in size, 30 packets/second can be sent in good conditions and 10 packets/second can be sent in bad conditions. RTT can vary depending on the user, therefore good and bad conditions can be defined in various different ways. However, it is safe to assume that an RTT of 250ms is a bad condition likely caused by a flooding of the connection, meaning the send rate needs to be slowed down.

Implementation:

- If current mode is “good” and conditions become bad, immediately switch to “bad” mode, slowing down send rate
- If current mode is “bad” and conditions have been good for a specific amount of time “t” then switch to “good” mode
- If the user drops from “good” mode back into “bad” mode in less than 10 seconds, double the amount of time “t” needed for conditions to be good before switching back into “good” mode. This should be clamped at a maximum value such as 60 seconds
- For every 10 seconds that passes in “good” mode, halve the time “t” needed to sustain good conditions before switching from “bad” to “good” mode again. Value should be clamped at a minimum, such as 1 second

This is a basic implementation of congestion avoidance but there are more advanced ways of doing this that incorporate many different factors such as packet loss percentage, network jitter and network type (LAN vs WAN) to better determine the rate at which packets can be sent.

2.4 Other things to consider

Whilst a lot of technical information has been covered surrounding architectures and software implementation, there is still a number of other things to consider that a developer could desire when using a networking component such as the one to be delivered in this project.

2.4.1 Peer-to-peer lockstep

This is a method of abstracting a game into a series of commands such as “move here”, “attack”, “construct building”, etc (Fiedler, 2010). This style of gameplay is common in RTS games and is relatively simple to network. To do so, simply communicate these commands with all the other players in the session, and have each player simulate those commands in their own instance of the game. However, to ensure that each instance of the game will play out the same, the turns cannot be simulated until all of the commands have been communicated. If this is not done, then the order of game commands could vary with some commands not even being simulated on some players devices. The problem with this though is that the player with the lowest ping in the session will negatively affect the experience for other players since everyone must wait for that player to finish communicating their turn. A programmer using this component may want to implement gameplay like this, so it is important to consider how this could be supported.

2.4.2 Client-side prediction

Latency is a big problem with client-server and dedicated server networking architectures due to the fact that the game’s simulation code is not being run locally, and instead the results are being communicated over a network. This can cause a noticeable amount of latency for players. When a player presses a key or begins a movement input, they need to wait for that input to be sent to the server, processed by the server before being sent back to the player, at which point they can update their local game code. The result of this delay is an appearance of unresponsiveness surrounding player inputs.

To mitigate this, some simulation code needs to be run client side. Specifically, the player simulation. Instead of just telling the server what it's doing, receiving game updates, and interpolating between updates. The client will now also predict the movement of the local player, ensuring that the player will see the results of their inputs immediately.

This isn't a flawless solution though, as now there is the question of what happens when the client gets different results surrounding the player movement compared to the server. The server should always have authority over clients, meaning, in the event of a disagreement of game updates, the client should always be corrected according to the servers' results. If the client had authority, it would be really easy for any player to cheat by allowing them to simply tell the server where they are and what they are doing, even if that's not necessarily the case.

But once again, latency becomes a problem. Because now, if the player receives corrected data about where they are, that data will be outdated by the time it reaches them. If a player is moving forwards and the server corrected their position, they would quickly move backwards which can be a jarring experience. This can be solved by keeping a circular buffer of player inputs. If the server corrects the client, the inputs stored in the circular buffer that are older than the corrected state should be discarded, and the states between the corrected state to the current predicted state should be replayed, essentially rewinding the last few frames of character inputs. It is important to note that this correction should be a rare occurrence, since the client and the server are likely to get the same results most of the time. The main cause of correction would most commonly be something external to the player's inputs such as colliding with an object or if the player is trying to cheat.

3 Requirements

The initial project proposal detailed a networking library and a Windows 10 game to demonstrate the library as the two deliverables of the project, with the library being designed for the MonoGame framework with potential for expanding support to the Unity game engine.

Originally, the game produced was planned to have a menu that allows the player to select various networking features they wish to have demonstrated, such as creating and joining peer-to-peer sessions or host a P2P with authority session instead.

However, the project changed in scope during development, where it was decided that supporting peer-to-peer would be the main focus, instead of supporting multiple architectures. This meant that the game produced to demonstrate the library wouldn’t need to allow for the player to choose specific features to see, as they would all be able to be shown at once.

3.1 Product requirements

As previously mentioned, developers using the library shouldn’t be expected to fully understand networking. Therefore, the synchronisation of game states between clients must be handled automatically. This can be done through the use of attributes in C#, where the developer can mark specific variables in a “NetworkedGameObject” derived class that they wish to be synchronised. The library should then construct and send packets using data from these networked variables.

This packet construction and sending process must also be handled automatically, as well as any packet processing and receiving. The game developer using the library should never have to come face-to-face with any individual packet, especially never have to manually construct one. Therefore, construction, sending / receiving and sequencing should all be handled in the background with no need for interference unless the developer wishes to debug any issues believed to be originating from the library.

Connections and loss of connection should all be seamless, with the developer being able to implement their own game specific behaviours for these events such as displaying a UI element or constructing an object representing a remote player.

Diagnostic information for each active connection should be accessible by the developer, this should cover the percentage of packets lost from the 100 most recently sent packets, the RTT (round trip time) of a connection and an estimated latency of the connection based on the RTT.

3.1.1 Performance Levels

Performance in games in something that people take very seriously, especially in competitive games where performance can provide advantages over other players. The popular tech focussed YouTube channel “Linus Tech Tips” investigated this idea by having players of various skill levels, from casuals to professional eSports players, play a competitive shooter at different framerates. Their individual performance was measured, and Linus Tech Tips (2019) found that “60fps at 60hz clearly had the worst results in every test” and “the consistency between runs was usually much better at 240hz”. That’s why it is important that this library doesn’t make a negative impact on performance. There are many ways the performance impact could scale, as more “NetworkedGameObjects” and variables are introduced, packets will become larger and the processes to create those packets could become more cumbersome.

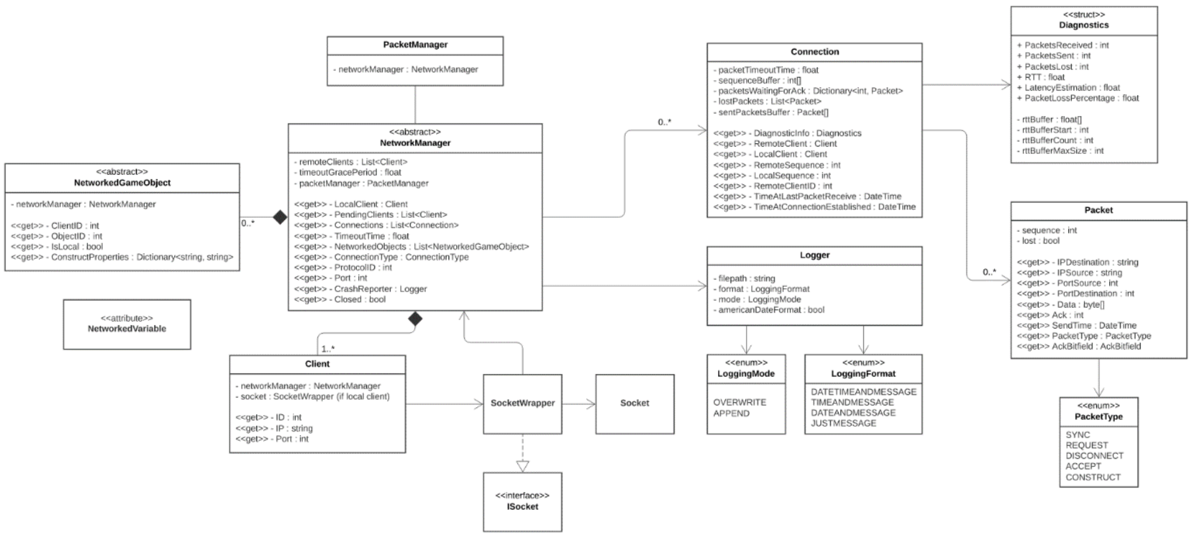
Therefore, comparisons will be made to measure the difference in performance between a singleplayer session and a networked session of the same game, with more comparisons being made as more players join the session. The metric to be measured here would be frames-per-second (fps). The framerates used in the Linus Tech Tips video were 60, 144 and 240, with the findings being that 60fps was a noticeable downgrade compared to 240, therefore dropping below 144 could be considered bad. The percentage decrease between 240 and 144 is 40%, so it is important that the performance doesn't degrade more than 40% during the performance tests to make sure that the implementation is suitable for online gaming and the expectations consumers have of it.

4 Design

4.1 Software design

4.1.1 Overview

There are three main elements of the library that the game developer will be interacting with: the NetworkManager, NetworkedGameObjects and the NetworkedVariable attribute.



PacketManager
- networkManager : NetworkManager
<<abstract>> NetworkManager
- remoteClients : List<Client>
- timeoutGracePeriod : float
- packetManager : PacketManager
<<get>> - LocalClient : Client
<<get>> - PendingClients : List<Client>
<<get>> - Connections : List<Connection>
<<get>> - TimeoutTime : float
<<get>> - NetworkedObjects : List<NetworkedGameObject>
<<get>> - ConnectionType : ConnectionType
<<get>> - ProtocolID : int
<<get>> - Port : int
<<get>> - CrashReporter : Logger
<<get>> - Closed : bool
<<abstract>> NetworkedGameObject
- networkManager : NetworkManager
<<get>> - ClientID : int
<<get>> - ObjectID : int
<<get>> - IsLocal : bool
<<get>> - ConstructProperties : Dictionary<string, string>
0..*
<<attribute>> NetworkedVariable
1..*
Client
- networkManager : NetworkManager
- socket : SocketWrapper (if local client)
<<get>> - ID : int
<<get>> - IP : string
<<get>> - Port : int
SocketWrapper
Socket
<<interface>> ISocket
0..*
Connection
- packetTimeoutTime : float
- sequenceBuffer : int[]
- packetsWaitingForAck : Dictionary<int, Packet>
- lostPackets : List<Packet>
- sentPacketsBuffer : Packet[]
<<get>> - DiagnosticInfo : Diagnostics
<<get>> - RemoteClient : Client
<<get>> - LocalClient : Client
<<get>> - RemoteSequence : int
<<get>> - LocalSequence : int
<<get>> - RemoteClientID : int
<<get>> - TimeAtLastPacketReceive : DateTime
<<get>> - TimeAtConnectionEstablished : DateTime
<<struct>> Diagnostics
+ PacketsReceived : int
+ PacketsSent : int
+ PacketsLost : int
+ RTT : float
+ LatencyEstimation : float
+ PacketLossPercentage : float
- rttBuffer : float[]
- rttBufferStart : int
- rttBufferCount : int
- rttBufferMaxSize : int
0..*
Packet
- sequence : int
- lost : bool
<<get>> - IPDestination : string
<<get>> - IPSource : string
<<get>> - PortSource : int
<<get>> - PortDestination : int
<<get>> - Data : byte[]
<<get>> Ack : int
<<get>> SendTime : DateTime
<<get>> PacketType : PacketType
<<get>> AckBitfield : AckBitfield
Logger
- filepath : string
- format : LoggingFormat
- mode : LoggingMode
- americanDateFormat : bool
<<enum>> LoggingMode
OVERWRITE
APPEND
<<enum>> LoggingFormat
DATETIMEANDMESSAGE
TIMEANDMESSAGE
DATEANDMESSAGE
JUSTMESSAGE
<<enum>> PacketType
SYNC
REQUEST
DISCONNECT
ACCEPT
CONSTRUCT

Figure 4.1: NetworkingLibrary UML class diagram

Figure 4.1 shows the relationships between all the classes in the library, with the centre of it all being the NetworkManager. This is in abstract class, meaning the game developer would need to create their own, game specific network manager that derives from this one to inherit all of its methods and data. In doing so, the developer now has the ability to interface with the library and implement their own custom behaviours upon certain events by overriding specific methods in the NetworkManager.

The NetworkManager holds references to 0 or more NetworkedGameObjects, which is another abstract class the game developer will use to specify which of their game objects they wish to have synchronised between clients over the network. Going one step deeper, the developer will then make use of the NetworkedVariable attribute to specify which variables of a networked game object they wish to synchronise.

4.1.2 Packet formats

Data will be sent over the network in the form of byte arrays. These byte arrays will be created from strings of data that vary between packet type:

- REQUEST = “[protocolID]/REQUEST/id=[clientID]”

- ACCEPT = “[protocolID]/ACCEPT/id=[clientID]/yourID=[remoteClientID]/ connectionNum=[numberOfConnections]/ connection1IP=[connection1RemoteIP]/ connection1Port=[connection1RemotePort]/…”

- SYNC = “[dataLength]/[protocolID]/SYNC/[timestamp]/[localSequence]/[remoteSequence]/id=[clientID]/objID=[objID]/VARSTART/[varName]=[varValue]/…/VAREND/”

- CONSTRUCT = “[dataLength]/[protocolID]/CONSTRUCT/[localSequence]/[remoteSequence]/id=[clientID]/objID=[objID]/[objType.FullName]/PROPSTART/[propName]=[propValue]/…/PROPEND/”

- DISCONNECT = “[protocolID]/DISCONNECT/id=[clientID]”

4.1.3 Establishing connections

In the literature review, it was mentioned that UDP is a connectionless protocol, so a custom implementation of connections needs to be created for this library.

Referring back to figure 4.1, it is shown that the NetworkManager can hold references to 0 or more instances of the Connection class. This class records the flow of packets between the local and a remote client. With that being said, using a method in the NetworkManager class, the game developer can call the “RequestConnection” method in the Client class (specifically, the local client instance) to begin the handshaking process of establishing a connection.

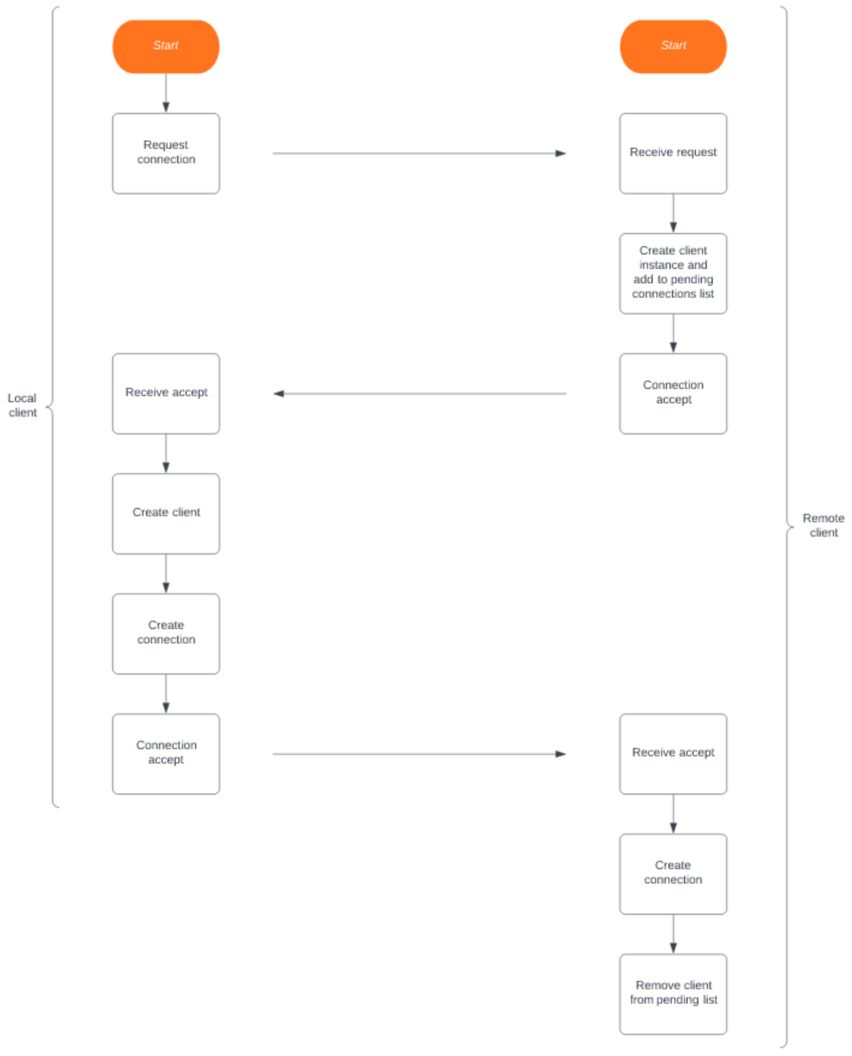


Figure 4.2: Connection establishment process

Figure 4.2 shows the local client (client 1) on the left and the remote client (client 2) on the right. To start the process, client 1 sends a connection request packet to client 2. Upon receive of this request, client 2 will process the packet (the algorithms used for packet processing will be detailed later in this document), create a client instance that represents client 1, add client 1 to a pending list and then send a connection accept packet back.

Client 1 receives this accept packet, processes it and creates a client instance representing client 2. It will also create an instance of a connection, which will store references to the local client and the remote client communicated with during this handshaking process. The final thing client 1 will do is send a connection accept packet back to client 2. After this point, client 1 will consider the connection established. As was detailed in the literature review, a connection over UDP is simply the state of sending and receiving packets consistently, whilst a disconnection is the opposite (no longer receiving packets), a connection timeout. So, even if client 2 doesn't succeed in establish the connection on its end of the handshake, it doesn't cause any problems for client 1 because after the timeout period has passed, client 1 will detect a timeout and no longer consider itself connected.

When client 2 does receive the connection accept from client 1, it will create a connection much like client 1 did and it will also remove client 1 from the pending list. After this, both clients will consider themselves connected to each other and immediately begin sending sync or construct packets.

5 Implementation and testing

5.1 Implementation

The networking library has been developed with support for peer-to-peer game sessions, where each client is directly connected to every client in the session. Each client is then responsible for communicating local changes so that other clients can replicate those changes.

As well as handling the sending and receiving of packets, the library also automatically synchronises networked game objects and their networked variables, all the developer has to do is specify which of these variables they wish to synchronise, and the library will use reflection to do so.

“Reflection provides objects (of type Type) that describe assemblies, modules, and types. You can use reflection to dynamically create an instance of a type, bind the type to an existing object, or get the type from an existing object and invoke its methods or access its fields and properties.” (Wagner, 2023).

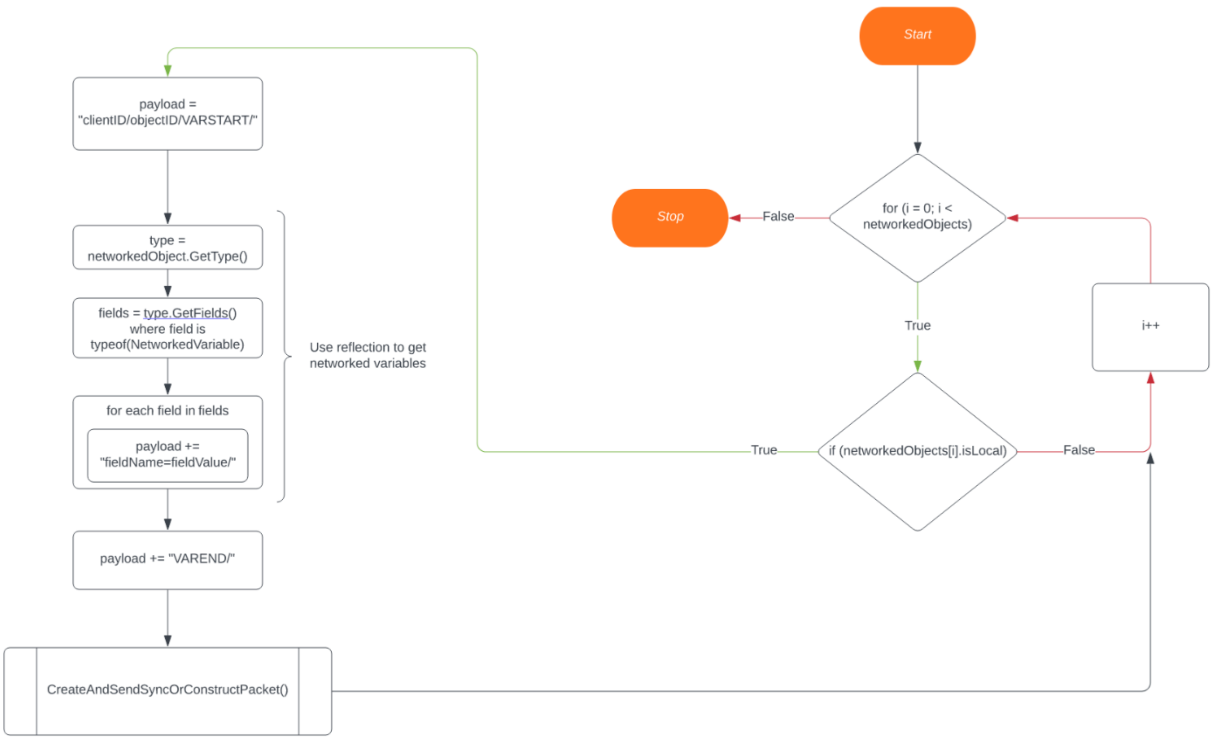
In the library, reflection is specifically used to find any fields marked with the NetworkedVariable attribute and get their values as well as finding variables with specific names and updating their values.

The following contents will explain some of the important algorithms used and decisions made.

5.1.1 Packet creation

5.1.1.1 Sync

One of the core methods of the network manager is “SendGameState” shown in figure 5.1. This is called in the Update method, which is a method the game developer should be calling every frame. This method is responsible for creating payloads for sync packets that will be sent over the network.



Start
for (i = 0; i < networkedObjects)
Stop
False
True
i++
if (networkedObjects[i].isLocal)
True
False
payload = "clientID/objectID/VARSTART/"
type = networkedObject.GetType()
fields = type.GetFields() where field is typeof(NetworkedVariable)
for each field in fields
payload += "fieldName=fieldValue/"
Use reflection to get networked variables
payload += "VAREND/"
CreateAndSendSyncOrConstructPacket()

Figure 5.1: SendGameState()

It loops over every networked game object in the manager, only proceeding with payload construction if the object is local. If this wasn't the case, clients would be attempting to communicate networked variables back to where they originate from where they have most likely already been changed locally.

If the object is local, payload construction begins. First, the clientID and objectID is added to the payload. Following this, reflection is used to retrieve every field in the object that is marked with the NetworkedVariable attribute. Once this is done, the payload will be appended with "VAREND" and the resulting string will be passed to another method which adds some more important data to the packet before it can be sent.

5.1.1.2 Construct

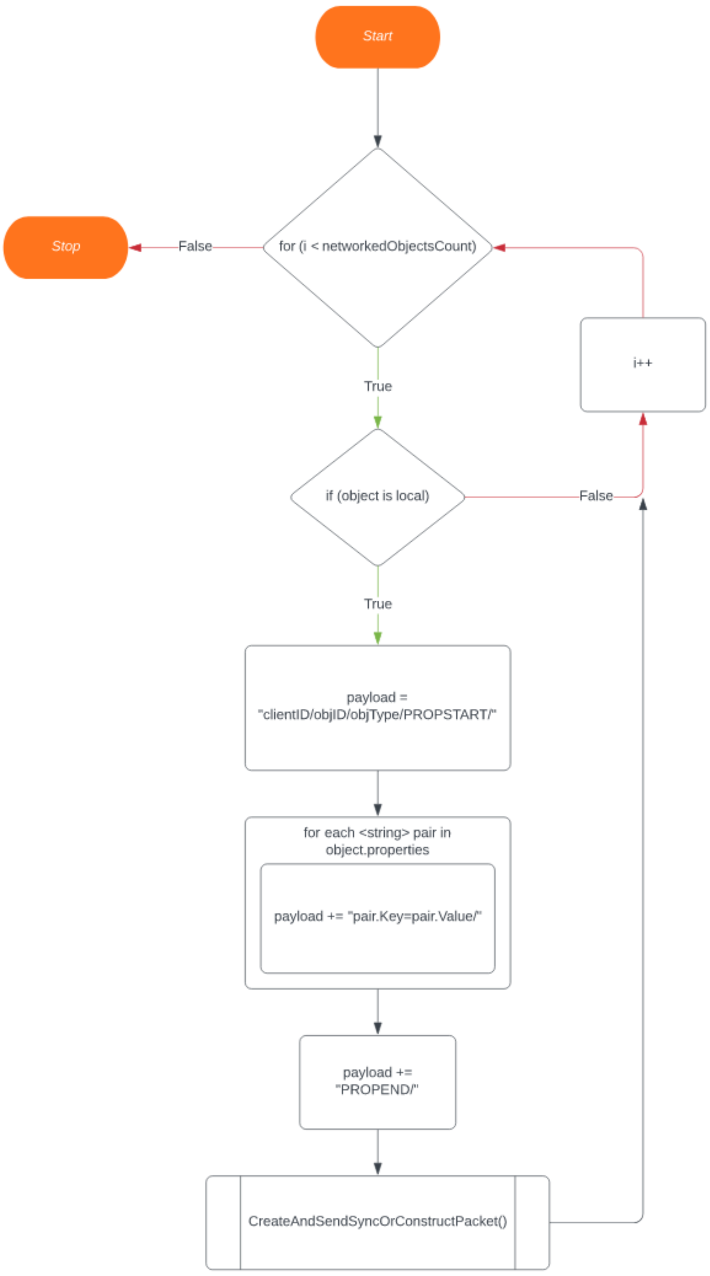

Start
for (i < networkedObjectsCount)
False
Stop
True
i++
if (object is local)
False
True
payload = "clientID/objID/objType/PROPSTART/"
for each <string> pair in object.properties
payload += "pair.Key=pair.Value/"
payload += "PROPEND/"
CreateAndSendSyncOrConstructPacket()

Figure 5.2: SendLocalObjects()

The SendLocalObjects method shown in figure 5.2 demonstrates how the payloads of construct packets are created. Again, for every locally created networked game object, a payload is created. Just like with sync packets, the clientID and objectID is used to identify the object associated with this packet. Then the payload is appended with any construction properties by looping through the construction properties for this object that were specified by the game developer.

5.1.1.3 Final construction before send

During the explanation of how sync and construct packets have their payloads created, it was mentioned that a method takes this payload and adds more data to the packet before it is sent. That is the “CreateAndSendSyncOrConstructPacket” method shown in figure 5.3.

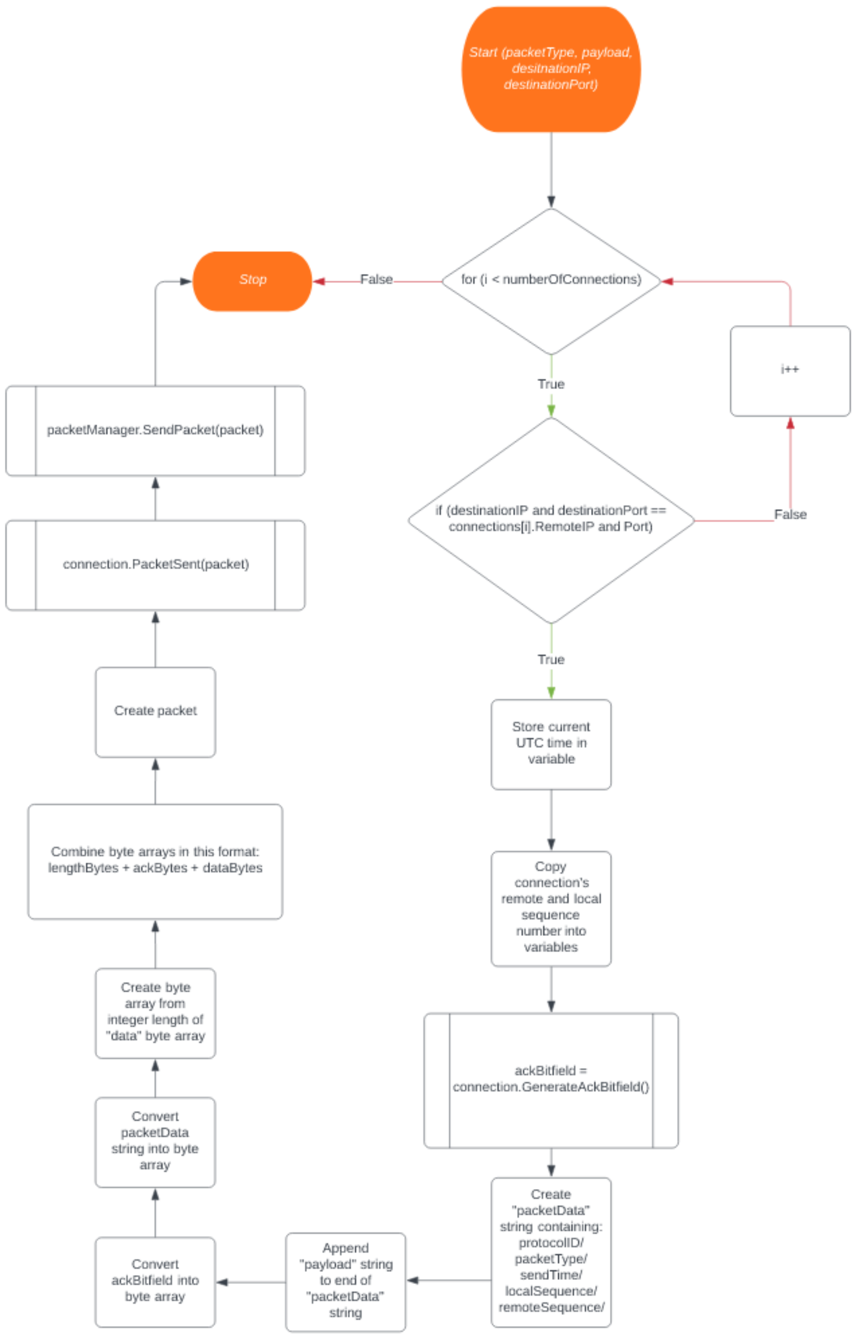

Start (packetType, payload, desitnationIP, destinationPort)
for (i < numberOfConnections)
False
Stop
True
i++
if (destinationIP and destinationPort == connections[i].RemoteIP and Port)
False
True
Store current UTC time in variable
Copy connection's remote and local sequence number into variables
ackBitfield = connection.GenerateAckBitfield()
Create "packetData" string containing: protocolID/ packetType/ sendTime/ localSequence/ remoteSequence/
Append "payload" string to end of "packetData" string
Convert ackBitfield into byte array
Convert packetData string into byte array
Create byte array from integer length of "data" byte array
Combine byte arrays in this format: lengthBytes + ackBytes + dataBytes
Create packet
connection.PacketSent(packet)
packetManager.SendPacket(packet)

Figure 5.3: CreateAndSendSyncOrConstructPacket()

This method will have received a payload and information about the destination connection, it also has the ability to specify that the destination is all connections, meaning the packet will be sent through all connections instead of just one.

Once the target connection is found, the current UTC converted time will be stored in a variable. Additionally, the local sequence and remote sequence values of the connection will be copied before generating an acknowledgement bitfield.

Once this data has all been gathered, a new string will be created that acts as packet header. It contains the protocolID, packet type, a timestamp and sequencing information. Following this, the previously created payload can be appended to the new string.

Now it's time to create the byte array that will be sent over the network; however, this isn't as simple as simply converting the string into bytes. The byte array that is sent must be a combination of multiple byte arrays.

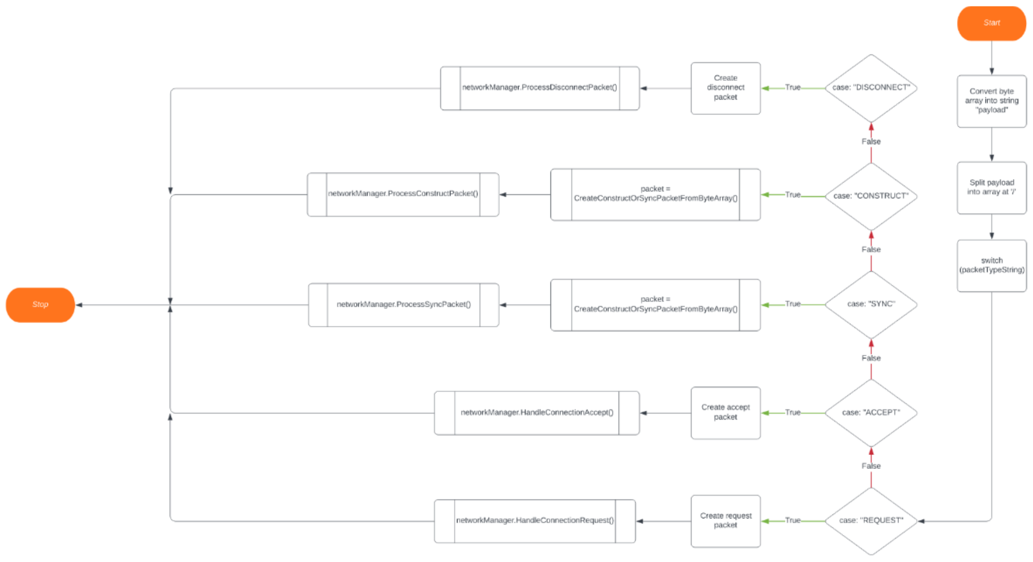
First, the ack bitfield must be converted into a byte array. This is important because this can't be included in the payload string. Once this is done, the packet data string will be converted into bytes. Finally, the length of this newly created data byte array will also be converted into bytes. This is used in packet processing so that the receiving client can separate the ack bitfield bytes from the data bytes.

Once all these byte arrays have been created, they are combined in this format:

- "lengthOfData/data/bitfield"

Now that the additional information has been added to the byte array, the final packet object can be created, where it will be passed to the connection to update sequencing and passed to the packet manager where it will be sent to the remote client.

5.1.2 Packet processing



Start
Convert byte array into string "payload"
Split payload into array at '/'
switch (packetTypeString)
case: "DISCONNECT"
True
Create disconnect packet
networkManager.ProcessDisconnectPacket()
False
case: "CONSTRUCT"
packet = CreateConstructOrSyncPacketFromByteArray()
networkManager.ProcessConstructPacket()
case: "SYNC"
networkManager.ProcessSyncPacket()
case: "ACCEPT"
Create accept packet
networkManager.HandleConnectionAccept()
case: "REQUEST"
Create request packet
networkManager.HandleConnectionRequest()
Stop

Figure 5.4: ConstructAndProcessPacketFromByteArray()

When a client receives a packet, it will be received by the PacketManager class, where the byte array will be converted into a string before being split at the separation character '/'. Now that the data has been split into separate strings, it can easily determine the type of packet for further processing using a switch statement as shown in figure 5.4.

5.1.2.1 Request

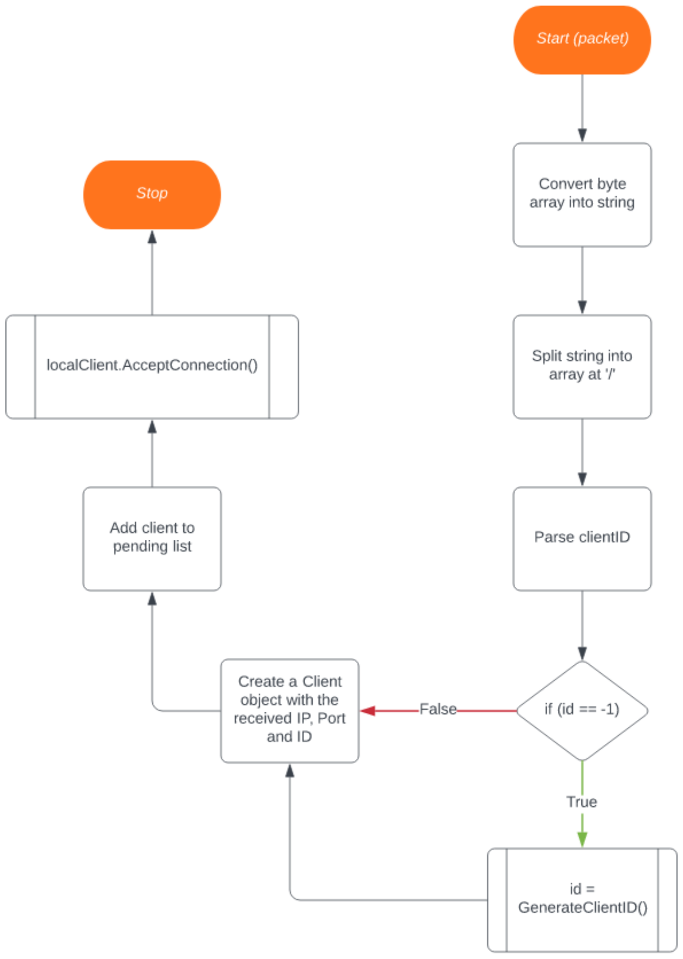

Start (packet)
Convert byte array into string
Split string into array at '/'
Parse clientID
if (id == -1)
True
id = GenerateClientID()
False
Create a Client object with the received IP, Port and ID
Add client to pending list
localClient.AcceptConnection()
Stop

Figure 5.5: HandleConnectionRequest()

If the packet manager determined an incoming byte array to be a request packet, it will pass it to the NetworkManager method “HandleConnectionRequest()” which can be seen in figure 5.5. First, it will convert the byte array into a string and, again, separate at the ‘/’ character.

The first variable to look at is the clientID. This value represents the ID of the sender. If the sender is not already in an existing session, its ID will be -1. So, if the clientID value of an incoming packet is -1, a new ID needs to be generated. This will be a randomly generated number between 100 and 999 and cannot be the same as any other IDs in the session as this will be used to determine which remote game objects originate from which clients.

With this information, a client instance will be created that represents the sender of the request packet using the incoming port, IP and clientID. Then, the client will be held in the pending list so that it can be referred to later in the handshake process. Following this, a connection accept packet is created and sent. During this creation process, an ID will also be generated for the local client if it does not already have one so that it can be included in the accept packet.

5.1.2.2 Accept

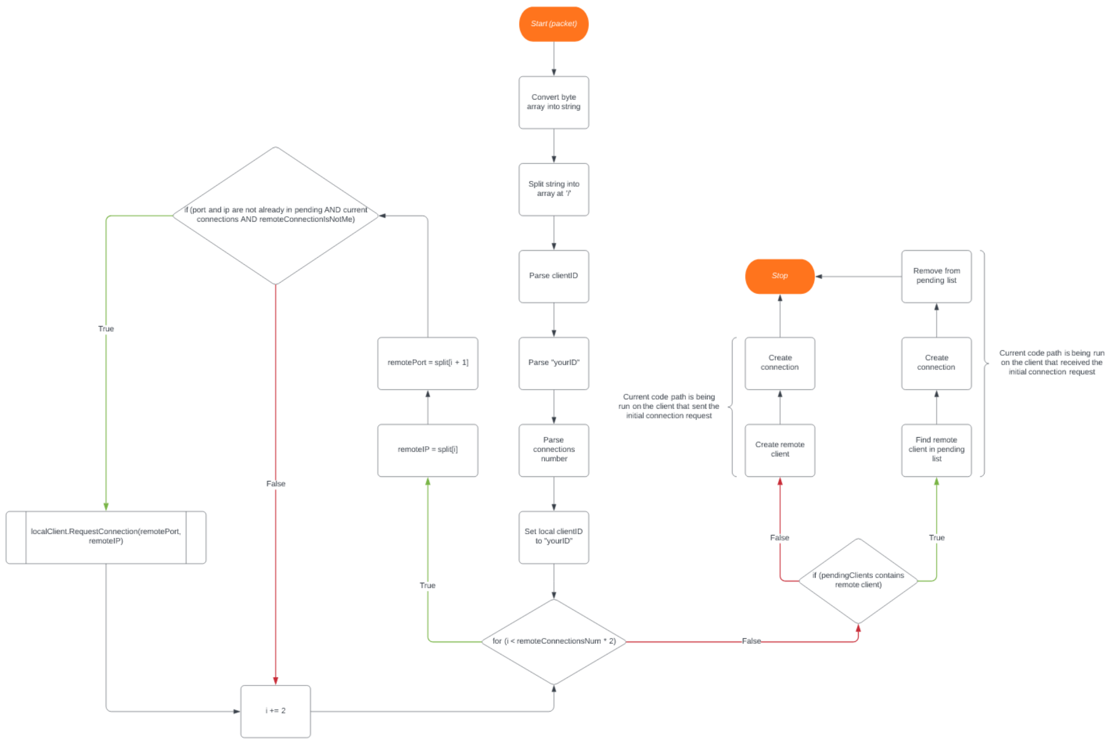

Start (packet)
Convert byte array into string
Split string into array at '/'
Parse clientID
Parse "yourID"
Parse connections number
Set local clientID to "yourID"
for (i < remoteConnectionsNum * 2)
True
remoteIP = split[i]
remotePort = split[i + 1]
if (port and ip are not already in pending AND current connections AND remoteConnectionIsNotMe)
True
localClient.RequestConnection(remotePort, remoteIP)
False
i += 2
False
if (pendingClients contains remote client)
False
Create remote client
Create connection
Current code path is being run on the client that sent the initial connection request
True
Find remote client in pending list
Create connection
Remove from pending list
Current code path is being run on the client that received the initial connection request
Stop

Figure 5.6: HandleConnectionAccept()

Accept packets have a bit more going on during processing because of many different factors. Firstly, this packet will contain a value called “yourID”. If the client receiving this packet didn’t already have any existing connections, then their clientID will be -1. The client they are attempting to establish a connection with sent this accept packet and generated an ID for them, this is the “yourID” value. This client will read this value and update their own ID with the new value. This is important because if clients generated their own ID before connecting to anyone, then there is a chance that the ID they generated could be the same as another in the session, as they didn’t know to exclude that ID during generation. Instead, clients will be told their own ID upon joining a session to ensure that they don’t get the same ID as another client.

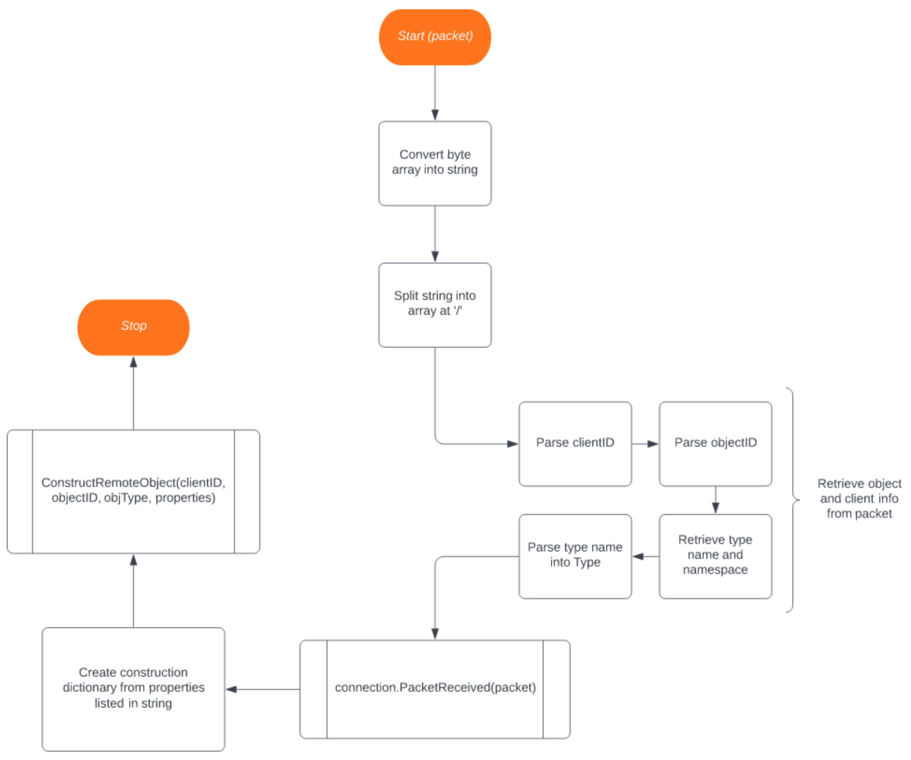
After this, existing connections will be copied. If the sender of this packet is already in a session with other clients, these connections will be detailed here, with each IP and Port number of each connection being listed. This client will simply loop through these fields and send a connection request to each client listed.

Once this is done, the path chosen is dependent on which client sent the initial connection request of the handshake detailed in figure 4.2. As mentioned before, the client that received the initial connection request will add the connecting client to a pending list. The sender of that initial request never adds the other client to the pending list. So, this means that if the sender of this accept packet exists in the pending list, then this code is being run on the client that received the initial connection request. In this case, that means that the client must be retrieved from the pending list so that a connection instance can be created. Once the connection is created, the client is removed from the pending list. On the other hand, if the sender of the accept does not exist in the pending list, then the code is being run on the client that sent the initial request. In this case, it will simply create a client and a connection before sending an accept packet back.

5.1.2.3 Disconnect

The disconnect packet is relatively simple. It only contains the ID of the sender and the keyword "DISCONNECT". From here, the NetworkManager will simply find the client associated with that ID and remove the connection to stop the sending of packets.

5.1.2.4 Construct



Start (packet)
Convert byte array into string
Split string into array at '/'
Parse clientID
Parse objectID
Retrieve type name and namespace
Parse type name into Type
Retrieve object and client info from packet
connection.PacketReceived(packet)
Create construction dictionary from properties listed in string
ConstructRemoteObject(clientID, objectID, objType, properties)
Stop

Figure 5.7: ProcessConstructPacket()

Construct packets are used to tell remote clients that the local client has constructed a networked game object and this needs to be replicated. The packet contains the clientID of the local client and the objectID of the game object.

Object IDs are generated in a similar way to clientIDs, except it can all be done locally without needing to know if a specific ID is already in use by another client. This is because the object ID is tied to a specific client ID so that when a sync or construct packet is processed, it will find the target object by searching for a specific object ID in a specific client.

The packet will also include the object type in its "full name" form, this includes the type and the namespace it came from. Before the packet is processed, the connection associated with the sender will be updated as having received a packet, what this does specifically will be detailed later.

5.1.2.4.1 Construction properties

The final thing included in this packet is a series of strings representing the construction properties of the object.

These are defined when the developer creates a local instance of a networked game object and are used to determine any parameters that may need to be used when a remote client constructs an instance of the object.

This is done by using a string dictionary, where the key is the name of the property, and the value is the value of the property.

For example, consider a game where multiple variations of a circle can be created at runtime depending on a user input and this circle needs to be replicated by another client. There are 3 colours (red, green or blue) the user can select from, and the user's choice determines which variation of the object gets created. If the user chose red, then a circle object would be created with an entry in the construction properties dictionary like this: "colour=red". This will then be included in the construct packet that gets sent over the network.

Once the library has retrieved the data it needs from the packet, it will call the abstract method "ConstructRemoteObject()" (which can be seen in figure 5.7) in the network manager, of which the developer should have implemented in their game specific network manager.

In this method, the developer will need to use an if statement to determine which parameters are used in the construction of the remote object depending on the properties included in the dictionary.

5.1.2.5 Sync

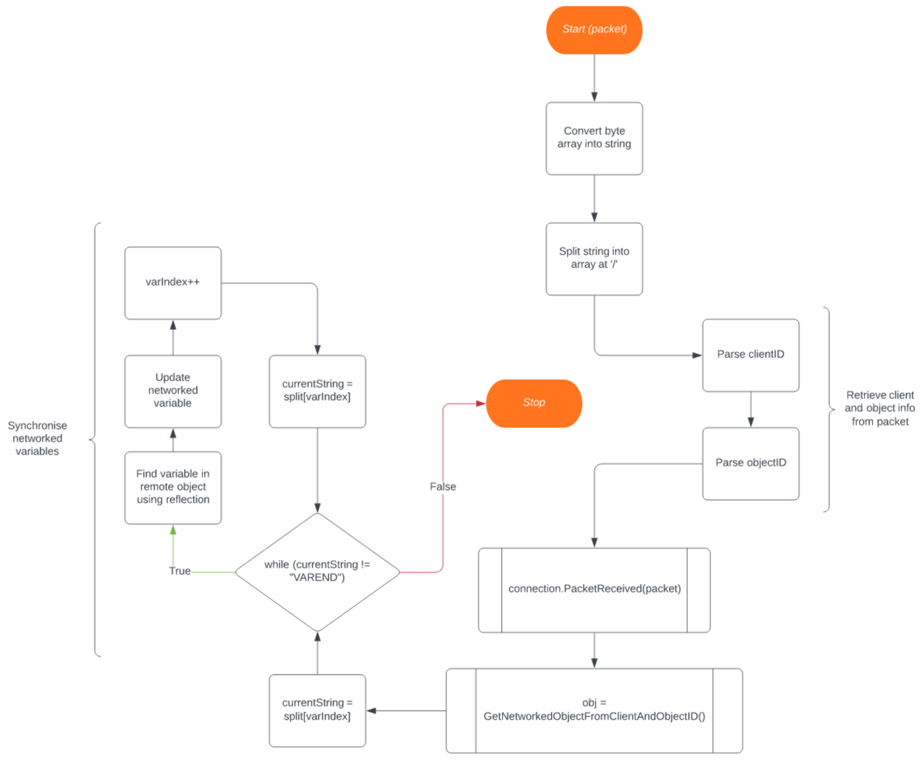

Start (packet)
Convert byte array into string
Split string into array at '/'
Parse clientID
Parse objectID
Retrieve client and object info from packet
connection.PacketReceived(packet)
obj = GetNetworkedObjectFromClientAndObjectID()
currentString = split[varIndex]
while (currentString != "VAREND")
False
Stop
True
Find variable in remote object using reflection
Update networked variable
varIndex++
currentString = split[varIndex]
Synchronise networked variables

Figure 5.8: ProcessSyncPacket()

When processing a sync packet, the remote object will be found by using the objectID paired with the clientID (GetNetworkedObjectFromClientAndObjectID in figure 5.8). Following these IDs will be a series of strings in a similar format to the construction dictionary. These represent networked variables.

To synchronise the remote object, these values are looped through, and reflection is used to find the variable with the name that matches the current string and update the value of that variable with the value that follows the current string. Once all networked variables are updated, the remote object is now synchronised.

5.1.3 Acknowledgements

5.1.3.1 How are connections updated?

As mentioned earlier, whenever a packet is sent or received, this packet is passed to the associated connection for some additional processing.

When a packet is sent, the “PacketSent” method of the connection is called (figure 5.9). This method will first ensure that the packet it was passed is a construct or sync packet. If it is not either of these types, it will simply ignore the packet as they are not included in any acknowledgement processes. When the correct packet type is passed to this method, the first thing it will do is increment the local sequence number. The local sequence number represents the number of packets sent, and this needs to be unique to each connection, so it is only incremented when a packet has been sent through that specific connection. Following this, the packet is added to the “WaitingForAck” dictionary, the key of this dictionary entry is equal to the sequence number of the packet. The reason a dictionary was used for this is because it is quicker to search for a specific entry in a dictionary in comparison to looping through a list of packets, checking the sequence number of each one and stopping when there is match. Then the packet is added to a circular buffer of 100 packets, representing the 100 most recently sent packets. This buffer is used in the calculation of packet loss percentage, which is the final method call in this process.

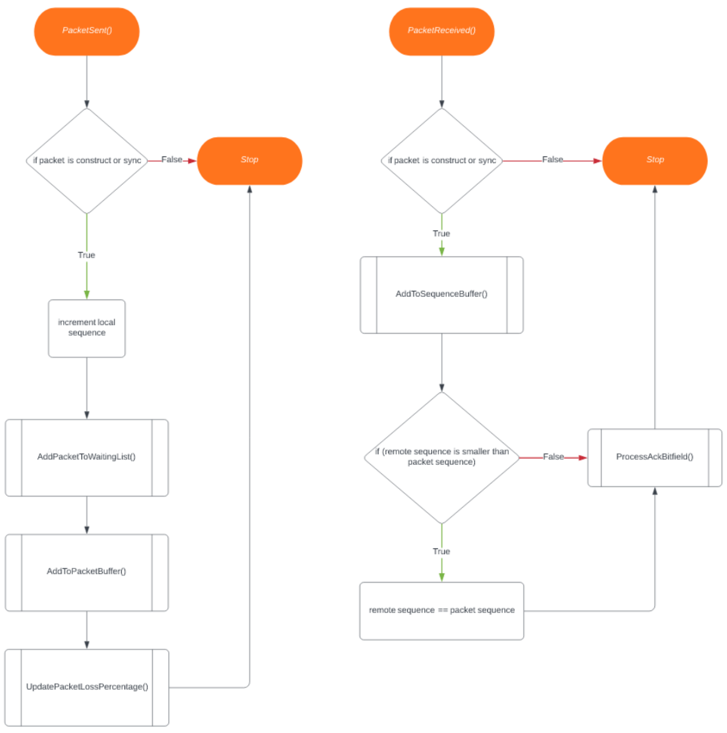

PacketSent()
if packet is construct or sync
False
Stop
True
increment local sequence
AddPacketToWaitingList()
AddToPacketBuffer()
UpdatePacketLossPercentage()
PacketReceived()
if packet is construct or sync
False
Stop
True
AddToSequenceBuffer()
if (remote sequence is smaller than packet sequence)
False
ProcessAckBitfield()
True
remote sequence == packet sequence

Figure 5.9: PacketSent() and PacketReceived()

When a packet is received, the “PacketReceived” method is called (figure 5.9). Much like for packet sent, this method only cares about the packet if it is a construct or sync packet. The first thing it will do is add the sequence number of the packet to another circular buffer “sequenceBuffer”. This buffer is limited to a size of 33, as in the 33 most recently received sequence numbers. If the received sequence number is higher than the current remote sequence number of the connection, then the remote sequence number will be updated with the higher value, if not, then this packet is older than the most recently received one. The final step in this process is to process the ack bitfield that was included in the received packet, but before this is explained, how does an ack bitfield get generated?

5.1.3.2 Generating the bitfield on send

The ack bitfield is a series of acknowledgements that specifies which of the 33 most recent packets have been received. It is generated by running the for loop “i < 33”. It will take the remote sequence number (most recently received sequence number) and check to see if this sequence – i (the current loop iteration) is included in the sequence buffer. If so, then this sequence needs to be acknowledged which is done by setting the corresponding bit in the bitfield.

5.1.3.3 Processing the bitfield on receive

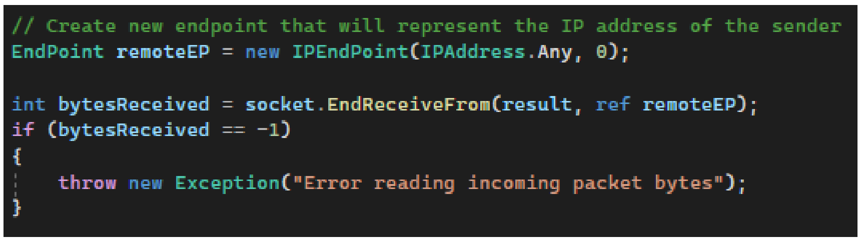
Processing the bitfield essentially works in the reverse of its creation. When a packet is received, it will include an acknowledgement as well as the acknowledgement bitfield. This acknowledgement is equal to the sequence number the sender most recently received. So, the bitfield is then scanned using the acknowledgement. If the first bit in the field is set, then that means the bitfield is acknowledging “ack - 1”. The waiting for ack dictionary will then be searched for this sequence number, if it exists in the waiting list, it will be removed as it has now been acknowledged.

Much like connection timeouts, packets can be determined lost if a certain amount of time has passed without receiving an acknowledgement for it. If this happens, the packet will be removed from the waiting list and then added to the lost packets list.

5.1.4 The Socket class

The networking library makes use of the .NET Socket class to transmit data. The socket begins receiving from any IP address and port with the asynchronous "BeginReceiveFrom" method. This means that the program can continue executing tasks while the socket waits for an input, when the socket does receive an input (byte array over the network) it will invoke the callback method. This callback is called "ReceiveCallback" in the packet manager.

Inside of this callback, the IP and port number of the sender needs to be retrieved (figure 5.10). This is done using the EndReceiveFrom method in the socket class.



```
// Create new endpoint that will represent the IP address of the sender
EndPoint remoteEP = new IPEndPoint(IPAddress.Any, 0);

int bytesReceived = socket.EndReceiveFrom(result, ref remoteEP);
if (bytesReceived == -1)
{
    throw new Exception("Error reading incoming packet bytes");
}
```

Figure 5.10: Obtaining the remote endpoint

First, an EndPoint is created with "any" IP address, meaning any IP address is allowed in this EndPoint instance, and a port number of 0, which means any port number is allowed. Then, the socket will end the receive and store the number of bytes in an integer. However, a great benefit of this end receive method is the "out" parameter. It can be used to retrieve the remote end point of the receive, i.e., the IP address and port number of the sender. So now the previously created remoteEP variable has been updated with the information of the sender of the most recently received packet, with this, the packet can be processed further.

The byte array is converted into a string so that the protocolID can be read. First, the IP address will be checked against the current connections, if it exists in the connections, then this packet should be processed. If not, then a final check is made against the protocolID. If this ID is the same as the protocolID used by this game, then the packet should be processed. If neither of these are true, the packet will be ignored.

Finally, the StartReceiving method will be called again to continue listening for incoming packets.

5.1.5 Threading

A problem was discovered during the late stages of development surrounding the packets waiting for acknowledgement dictionary in the Connection class and the connections list in the NetworkManager class. Both of these collections are iterated through in the code.

The rest of the library didn't use threading anywhere, instead using the asynchronous methods in the socket class. As previously mentioned, these methods will invoke callback methods when they receive data, allowing the code to continue executing whilst waiting for this response. So, the problem lies with the question: "What happens then, if the socket invokes a callback which adds a connection to the connection list… but, just before the callback was invoked, the NetworkManager was looping through the Connections list?". This would cause an exception to be thrown as the collection was modified before the loop could exit.

To fix this issue, threads and locking was used, specifically on the two collections mentioned. A thread is launched whenever the packetsWaitingForAck dictionary is modified, and this modification takes place in a locked section of code. This critical section of code will block any other threads from entering it until the current thread has exited this section, fixing the issue of modified collection exceptions.

There is a performance impact with thread locking, specifically if the critical section of code is very complex and takes a long time to execute which can slow down the program as a whole if multiple threads are sat waiting to enter this section. In this library, the critical section is relatively simple, but it is important to acknowledge the impact this had which could have affected the performance tests detailed later in this document.

5.1.5.1 Why does the socket use an interface?

The socket class is also created slightly different to how it would normally be used. It is instead stored in the SocketWrapper class, which uses the ISocket interface. This is because it allows the socket to now be mocked in testing. Without this interface, the socket can still be mocked but none of its methods can actually be overridden. Instead, using an interface allows for these methods to be overridden so that custom, test specific behaviours can be implemented so that a socket can be used in testing without actually having to send or receive data over the network.

5.1.6 How the game uses the library

5.1.6.1 Interacting with the game

The game created to demonstrate this library consists of some simple menus and a simple, 2D scene in which the player can choose their own player sprite and either create a networked session or join a session. When creating a session, the player must specify which port number they wish to use. When joining a session, the player must specify the exact IP address and port number they wish to connect to, as well as specifying the port number they wish to use locally.

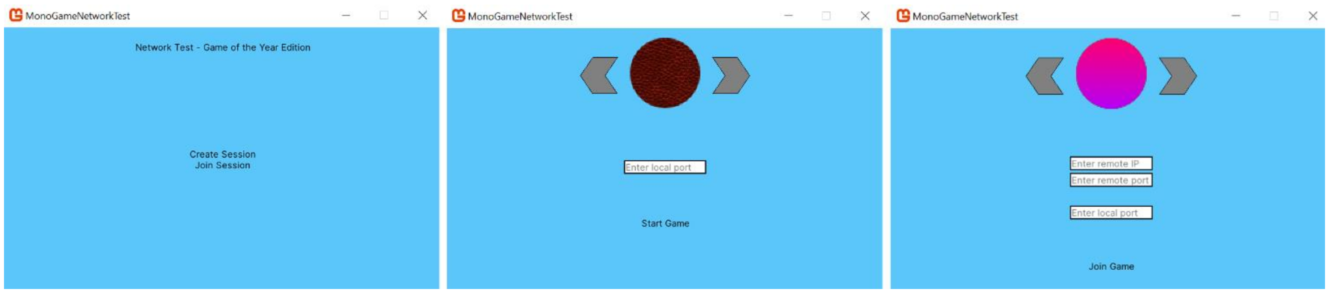


Figure 5.11: Game menus (left: main menu, middle: create session, right: join session)

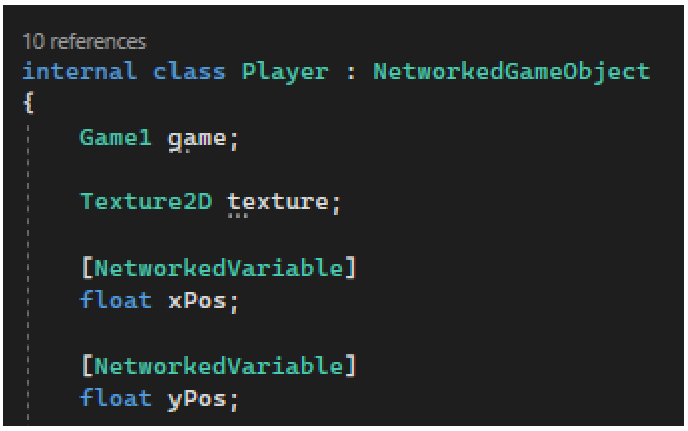
The reason for having the manual input of a port number shown in figure 5.11 was simply for testing purposes. It was originally planned to have the networking library default to port 27000. However, this caused some issues when trying to run multiple instances of the game on the same device, as multiple sockets would be trying to use the same port which caused crashes.

It was considered to have a more dynamic port selection, where a range of ports would be pre-selected, and the library would choose any available port from that selection, but the final decision was to just let the game developer specify which port they wish to use. In doing so, this allows the developer to expand on this in their own game code by potentially creating matchmaking systems specific to their game or a session browser that stores IP and port combinations of all existing sessions so that a player doesn't need to worry about it. If time were to allow, this could have been implemented into the library, but other areas of development required more focus, therefore it was suitable to just have a manual port and IP selection for demonstration purposes.

To create these menus and UI elements, a third-party library was used called "Myra". It is a UI library developed specifically for MonoGame and was used to create the previously mentioned menus as well as the UI labels that appear in-game. Another feature of Myra is the external UI editor called "MyraPad" which allows a user to create and organise UI elements and export them as a C# class. In the game project, this can be seen with the class names appended with ".Generated". This library is published under the MIT license (Shapiro, 2020).

5.1.6.2 Network manager and networked game objects

The networking library's NetworkManager is an abstract class that needs to have a game specific network manager derive from it. This is called the "Game_NetworkManager" in the game project. There are two networked game objects used in this demonstration, the player and the bullet.



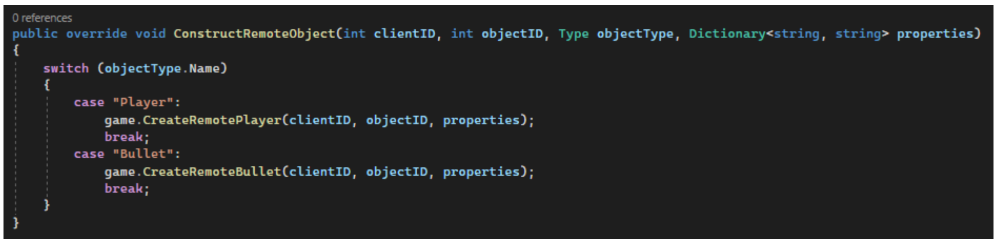
```
10 references
internal class Player : NetworkedGameObject
{
    Game1 game;

    Texture2D texture;

    [NetworkedVariable]
    float xPos;

    [NetworkedVariable]
    float yPos;
```

Figure 5.12: "Player" networked game object

As seen in figure 5.12, the Player class has two networked variables, xPos and yPos, both being float types. If this player is local, then it can be controlled using the arrow keys. These inputs will change the xPos and yPos values accordingly, with those changes being communicated over the network by the networking library.

5.1.6.3 Constructing a remote object

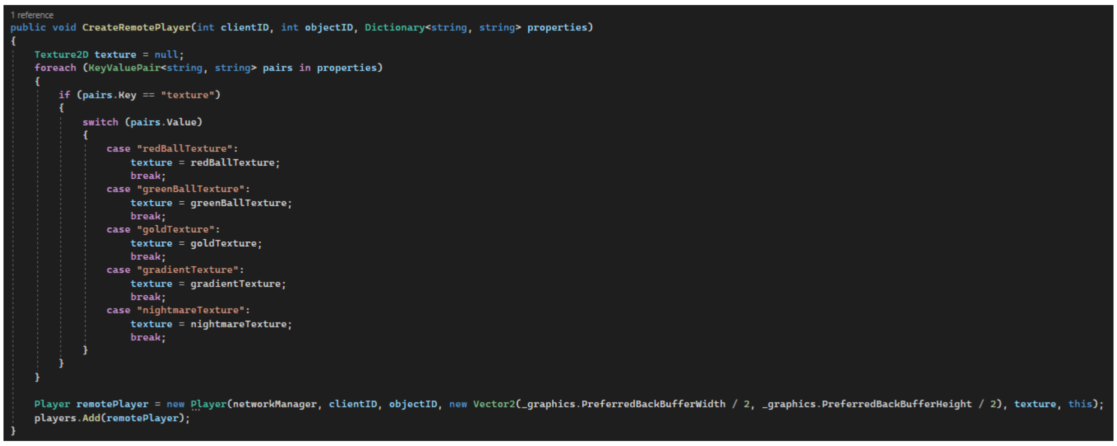
The abstract method "ConstructRemoteObject" is called by the network manager when a construct packet is received. It requires the game developer to implement a game specific reaction to this method call in their own network manager.



```
0 references
public override void ConstructRemoteObject(int clientID, int objectID, Type objectType, Dictionary<string, string> properties)
{
    switch (objectType.Name)
    {
        case "Player":
            game.CreateRemotePlayer(clientID, objectID, properties);
            break;
        case "Bullet":
            game.CreateRemoteBullet(clientID, objectID, properties);
            break;
    }
}
```

Figure 5.13: ConstructRemoteObject()

Figure 5.13 shows an example of how this can be approached. A simple switch statement will determine which remote object creation method should be used. If the object type was "Player" then the CreateRemotePlayer method will be called in the game class.



```
1 reference
public void CreateRemotePlayer(int clientID, int objectID, Dictionary<string, string> properties)
{
    Texture2D texture = null;
    foreach (KeyValuePair<string, string> pairs in properties)
    {
        if (pairs.Key == "texture")
        {
            switch (pairs.Value)
            {
                case "redBallTexture":
                    texture = redBallTexture;
                    break;
                case "greenBallTexture":
                    texture = greenBallTexture;
                    break;
                case "goldTexture":
                    texture = goldTexture;
                    break;
                case "gradientTexture":
                    texture = gradientTexture;
                    break;
                case "nightmareTexture":
                    texture = nightmareTexture;
                    break;
            }
        }
    }

    Player remotePlayer = new Player(networkManager, clientID, objectID, new Vector2(_graphics.PreferredBackBufferWidth / 2, _graphics.PreferredBackBufferHeight / 2), texture, this);
    players.Add(remotePlayer);
}
```

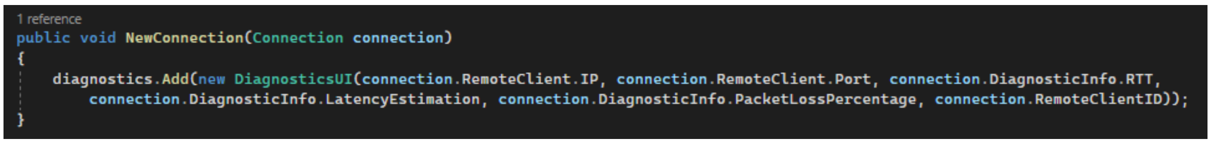
Figure 5.14: CreateRemotePlayer()

Going one step further then, this is where the construction properties dictionary that was detailed earlier becomes important. In the example shown in figure 5.14, the one variable that can change between players (and isn't directly serializable to and from a string) is the texture. This field of the player class uses the MonoGame Texture2D type and cannot be included in the byte array. So, when the local player was created, it was created with the texture that the player chose in the menu. Depending on which selection they made, a key/value pair was added to the construction properties dictionary. For example, if they chose the red ball as their player sprite, the entry added to the dictionary would look like this: "texture=redBallTexture". This is the string that is included in the construct packet.

So, when this construct packet is received, a null texture is created. Each key/value pair in the dictionary is then looped through (in this particular example, there will only be one pair so a loop isn't exactly necessary, but it demonstrates how this would work for more than one property). It finds the key called "texture" and a switch statement is used to determine which texture should be used for the remote player, filling in the null texture which can be passed to the player constructor.

5.1.6.4 Connect, disconnect and timeout events

When a connection is established, the ConnectionEstablished method will be called in the game specific network manager. In this method, the SendLocalObjects method of the network manager is called targeting the newly created connection. Another method is called specific to this game that updates the connections UI with a new diagnostics label (figure 5.15) which makes use of the Diagnostics struct associated with each Connection in the library.



```
1 reference
public void NewConnection(Connection connection)
{
    diagnostics.Add(new DiagnosticsUI(connection.RemoteClient.IP, connection.RemoteClient.Port, connection.DiagnosticInfo.RTT,
        connection.DiagnosticInfo.LatencyEstimation, connection.DiagnosticInfo.PacketLossPercentage, connection.RemoteClientID));
}
```

Figure 5.15: Adding a new diagnostics UI label

When a client disconnects or times out, the game also implements methods to remove the networked objects associated with the lost connection as well as removing the connection diagnostics label.

5.1.7 Unit testing

Unit testing has many benefits regarding software development. For this project, they serve two purposes. First being test driven development, the process of writing automated tests before developing the feature the test is written for. The test is written according to what the expected outcome of a planned feature is, before implementing the feature and making it pass.

Since this software wasn't fully completed and some features had to be cut, these unit tests provide another benefit. Any developer looking to extend the library with more features or refine existing ones will be able to run the tests to make sure that any changes made do not break existing systems.

These tests needed to cover all the important aspects of the library. In doing so, 41 tests were written which can be broken into several categories:

- ConnectionEstablishmentTests - testing the handshake process of establishing connections under various conditions
- ConnectionTests – testing methods in the Connection class such as checking for lost packets or generating acknowledgement bitfields
- DiagnosticsTests – testing the updates of the diagnostic values for connections
- LoggerTests – testing the various formats and modes of the logger class
- NetworkManagerTests – tests that include checking for timeouts, sending game state and sending local objects among others
- PacketManagerTests – testing the behaviours of packet I/O as well as mocking the socket class
- PacketProcessingTests – testing how different packet types are processed

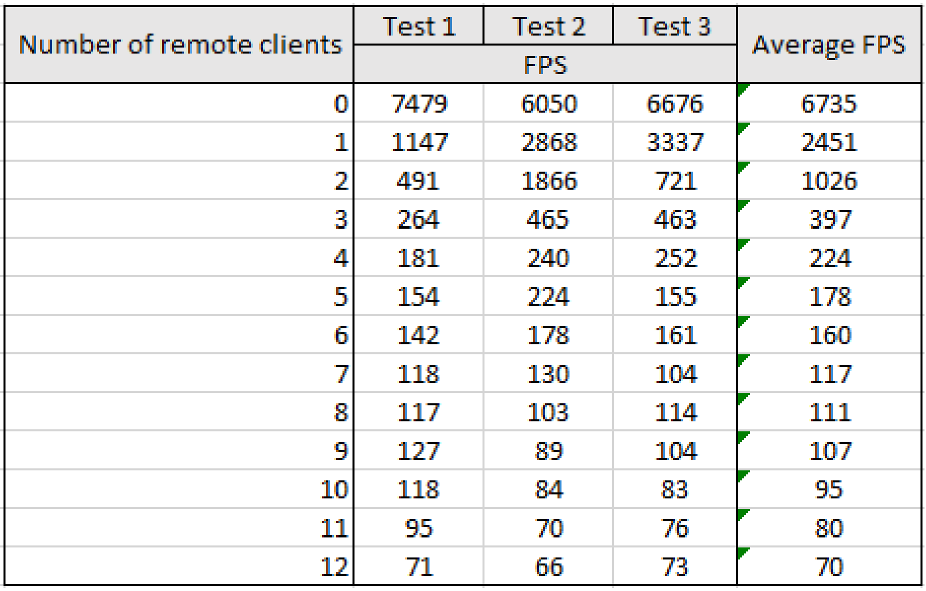
5.1.7.1 Choosing the testing framework

There are three, very popular and commonly used unit testing frameworks that were considered for this project: NUnit, MSTest and xUnit. They were experimented with briefly to determine which one should be used.

Whilst all three of them share a lot of similarities, it was decided that NUnit would be best for this project. It is very intuitive in its naming conventions, especially compared to xUnit, whilst also providing access to a lot more assertion types. MSTest was found to also be intuitive but lacked the assertion offering that NUnit had. Additionally, MSTest was also integrated directly into Visual Studio, but NUnit could also do the same thing by installing the NUnit Test Adapter package.

5.1.8 Performance tests

Performance of the library was tested by recording the change in fps (frames-per-second) after a new remote client joins the session. The client recording the changes was the only instance of the game running on the device, whilst the other clients were running on a separate laptop. If all clients were running on the same device, then the recorded changes wouldn't be accurate as the performance of each client would be degraded purely because they would all be sharing the same processing resources. The results of these tests can be seen in figures 5.16, 5.17, 5.18 and 5.19.



Number of remote clients	Test 1	Test 2	Test 3	Average FPS
	FPS			
0	7479	6050	6676	6735
1	1147	2868	3337	2451
2	491	1866	721	1026
3	264	465	463	397
4	181	240	252	224
5	154	224	155	178
6	142	178	161	160
7	118	130	104	117
8	117	103	114	111
9	127	89	104	107
10	118	84	83	95
11	95	70	76	80
12	71	66	73	70

Figure 5.16: Performance test results table

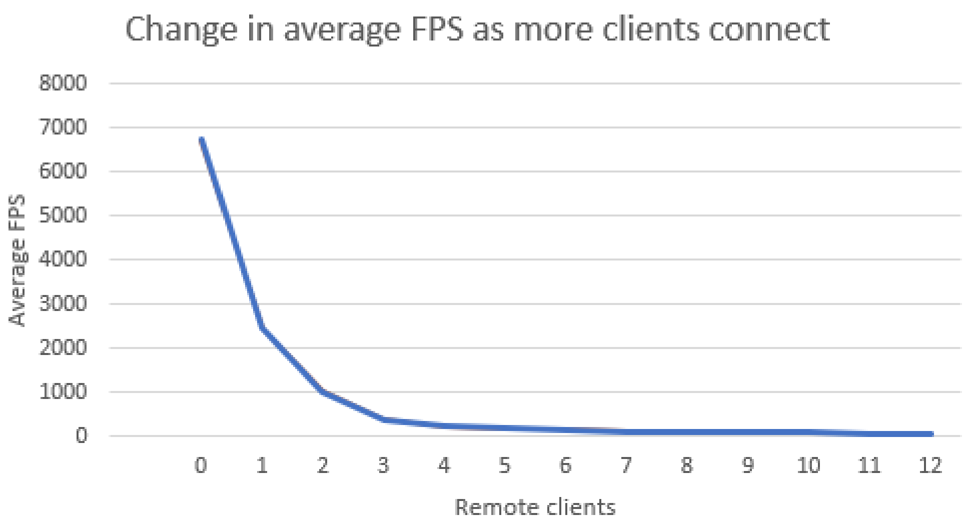


Figure 5.17: Change in average FPS as more clients connect (0 – 12)

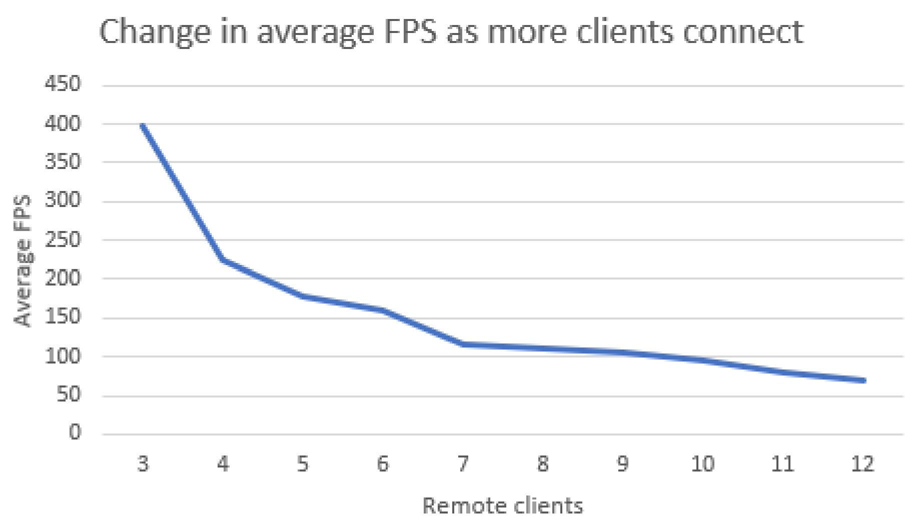


Figure 5.18: Change in average FPS as more clients connect (3 - 12)

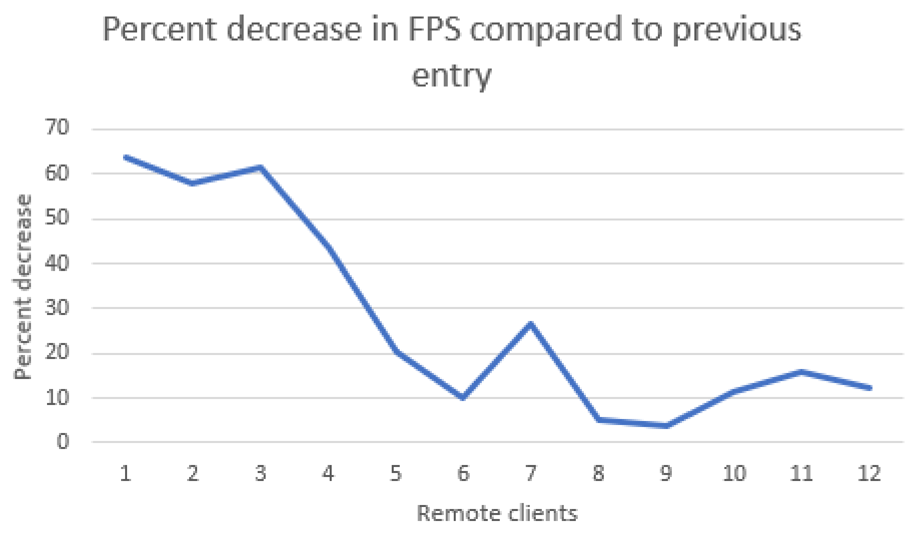


Figure 5.19: Percent decrease in average FPS compared to previous entry

6 Evaluation and discussion of results

6.1 Objective 1: “Develop a groundwork for the game using C# and MonoGame”

This objective was met by creating a simple 2D game which allowed a player to take control of a sprite in an empty scene. It also proved useful for experimentation during the design process. Instead of just creating a local scene, it was expanded upon with prototype networking code to experiment with ideas. This experimentation played a key role in identifying user stories.

6.2 Objective 2: “Create a reusable networking component as a DLL file that is capable of providing multiplayer features to an existing game”

This was the main deliverable of the project, and the objective was met. The outcome was a DLL file as specified and is able to provide a good range of networking features such as diagnostics, connection management and automatic synchronisation. However, some features had to be cut as the project changed in scope around the midpoint to include the use of unit testing.

6.3 Objective 3: “Create a prototype game to test the networking component”

This objective was met by expanding the game made in objective 1. Firstly, all of the prototype networking code from that original game was removed and relied solely on the networking library produced in objective 2. The resulting game allows players to choose between creating a session or joining an active one. Although, the player must manually input port numbers and IP addresses which isn’t ideal. However, the game does demonstrate all of the features of the library as was required by the objective. It allows players to choose from a selection of sprite textures which can be replicated appropriately by connected clients, demonstrating construct packets. It demonstrates synchronisation by replicating player movement inputs. It also further demonstrates construct packet types by allowing the player to fire a “bullet” by pressing the spacebar, instead of only constructing remote objects upon a new connection. Furthermore, the game demonstrates connection, disconnect and connection timeout events appropriately as well as displaying diagnostic information of each current connection in UI elements.

To further support the showcase of networked game objects vs local objects, a camera was introduced. This camera is fixed above the local players sprite at all times, meaning each client will have a different perspective of the game scene in comparison to others.

6.4 Objective 4: “Testing and evaluation of performance”

This objective detailed the use of unit testing and a performance evaluation.

First, the unit tests. These were a late addition to the project and, whilst they could have been done better, proved very beneficial to the project as a whole. For example, there was an issue with the ReceiveCallback method in the packet manager that would occur when an incoming packet didn’t contain a separation character in the string it was converted to. Before discovering this, the method just assumed all incoming packets would have this which isn’t the case. One of the key roles this method plays in the library is to determine whether or not an incoming packet even belongs to the game protocol, and if a rogue packet was to arrive in this method and it didn't contain the separation character when converted to a string, an exception would be thrown when an attempt is made to split the string around the non-existent separation character. Unit testing helped the discovery of this issue earlier than would have been the case had the bulk of testing been done manually at the end of the project.

Furthermore, as was mentioned earlier, unit tests open up the possibility for further work by third party developers. Since the library is not technically feature complete, missing some quality-of-life features such as matchmaking or packet flow control, a developer could choose to implement these features or refine existing ones all whilst using the current unit tests to ensure that any modifications made don't break any individual components of the library.

6.4.1 Performance evaluation

FPS (frames-per-second) changes were recorded as more clients joined the session. These changes were recorded over 3 separate test sessions and an average was taken, as seen in figure 5.16. Starting from 0 remote clients (base performance with no network activity) to 12 remote clients, quite the drastic impact can be seen as the game begins to manage more connections.

Going from 0 remote clients with an average of 6735 FPS, to 1 remote client with an average of 2451 FPS is a massive 63.6% decrease in performance. Then, again from 2451 to 1026 FPS with the addition of another client shows a 58% decrease from the performance of 1 remote client, and a performance decrease of 84.7% from the base performance of 0 remote clients. Going further, with a session of 12 remote clients, the performance of the local client dropped to 70 FPS, a staggering 98.9% decrease from the original base performance level.

Figure 5.19 shows the percentage decrease in FPS as each new client joins the session. The graph shows that there is a very sharp decrease in performance initially but seems to calm down as more clients are added, especially after the 7th remote client where the percentage decrease never goes back above 20 for the following results. It's hard to say exactly why the changes become less drastic like this, however, it is very clear that there are some design issues from the initial 63.6% decrease with the addition of the first remote client. This is especially bad when taking into account the fact that this is a prototype game that has no physics calculations and only negligible graphics calculations. How would this library affect a game that is already resource demanding to run even in local sessions? It's difficult to say without directly testing with a game like this, however, there are still some clear improvements that could be made.

6.5 The overall goal

The overall goal was to provide a networking library that any game developer could use with minimal effort and without needing to fully understand how networking works.

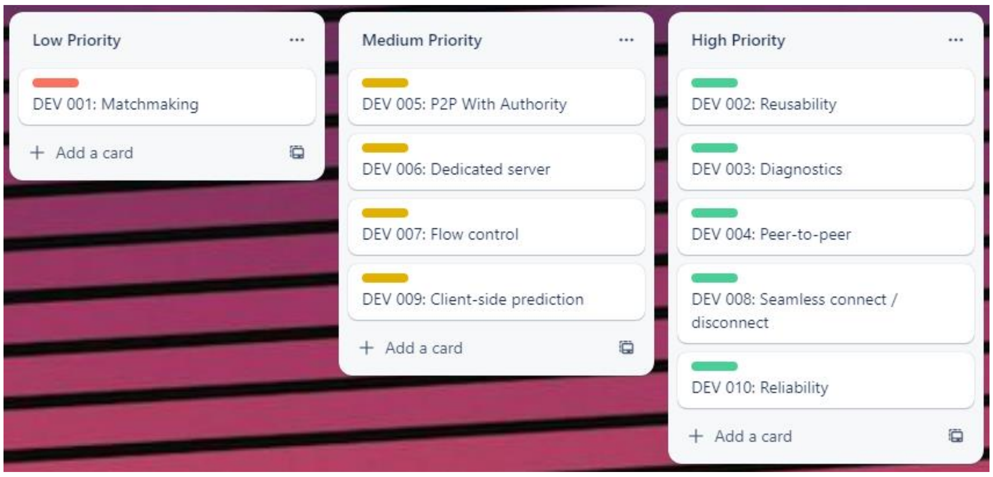
Firstly, the library has met the goal of being easy to use without previous experience or knowledge with networking. The setup of networked game objects and networked variables is clear and straightforward, and most of the synchronisation process is fully automatic, without needing interference from the developer.

However, one area of the synchronisation process was a lot more manual than initially hoped. The remote object construction. This requires the developer to define specific construction properties when creating a local instance of a networked object, so that these properties can be accounted for when the remote object construction method is called, of which the developer would need to implement a series of if statements or switch statements to retrieve the correct parameters when creating the remote object.

A solution was nearly found that allowed this process to also be fully automatic. The plan was to have the developer mark the remote constructor with a custom attribute like they would with a networked variable. Then, the library would find this remote constructor and read the parameter types and names, matching the names to variables that shared the exact name in the networked game object. For example, if the constructor took a float parameter called "health", the library would try to find a variable in the object called "health" and take its value much like it would with the networked variable attribute. Once all the parameters are retrieved, they can be sent in a construct packet. Upon receival of this construct packet, the remote constructor of the target object could be called, filling in the parameters from the packet. However, this only worked with "IConvertible" types, just like how the NetworkedVariable functionality only works with "IConvertible" types. Whilst it's fine to limit the variables to this, limiting remote object creation to these types isn't as feasible. During object creation, it is likely that they will take parameters that are not convertible, such as a Game class, or any other custom objects that are not convertible. So, it became a choice between two options, continue with the fully automatic approach and require that all networked game objects are only created with convertible types as parameters OR scrap the fully automatic solution in hopes that it provides developers with more freedom in how they create networked game objects. As shown in the implementation, the latter was chosen.

6.6 Project management

The chosen development methodology was agile. During the design phase, some user stories were identified and ranked based on priority which can be seen in figure 6.1.



Low Priority	Medium Priority	High Priority
DEV 001: Matchmaking	DEV 005: P2P With Authority	DEV 002: Reusability
	DEV 006: Dedicated server	DEV 003: Diagnostics
	DEV 007: Flow control	DEV 004: Peer-to-peer
	DEV 009: Client-side prediction	DEV 008: Seamless connect / disconnect
		DEV 010: Reliability

Figure 6.1: User stories priority ranking (full user stories breakdown in Appendix A.1)

The development was split into 4, 2-week sprints, with each sprint having 1 day for planning before beginning, and another day after finishing for reflection. Three stories were chosen for the first sprint: DEV 008, DEV 004, and DEV 003. In the retrospective, the developer had this to say about the first sprint:

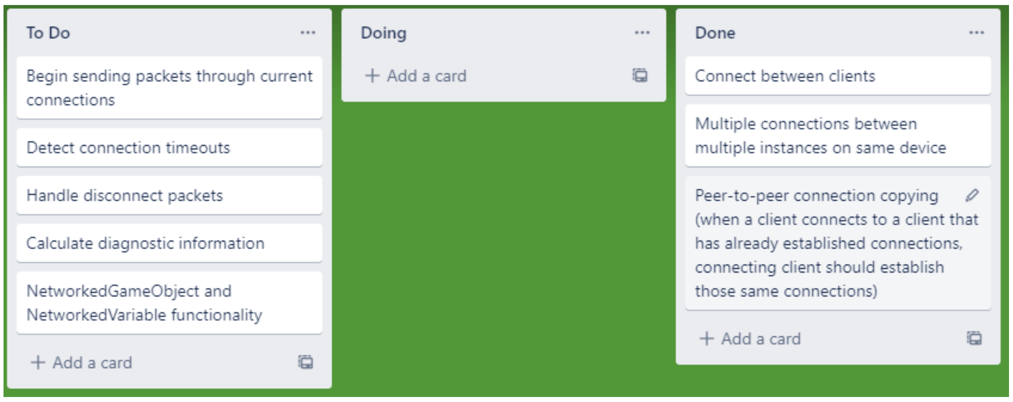
"Some progress was made, for example, the project is a lot clearer in its direction and how a game developer would use the final product."

However, also going on to say:

"…not enough progress was made. In the days I spent working on the project, not enough hours of those days were productive."

"I was always expecting the first sprint to prove quite difficult, and I think that expectation has been proven. The project plan makes it clear that the first development milestone is peer to peer support, which was planned to be achieved at the end of sprint 2. I planned it this way because I knew the first sprint would be challenging, which may be the reason for the lack of progress. However, now that the work has been done, I think the work will go a lot smoother from now on, as more pieces begin to fall into place."

No stories were completed at the end of sprint 1, indicating they were too big and much more akin to epics. However, these user stories were broken up into tasks, which were placed in a separate board to the user stories. At the end of sprint 1, 3 smaller tasks were completed as shown in figure 6.2.


To Do

- Begin sending packets through current connections
- Detect connection timeouts
- Handle disconnect packets
- Calculate diagnostic information
- NetworkedGameObject and NetworkedVariable functionality

+ Add a card

Doing

+ Add a card

Done

- Connect between clients
- Multiple connections between multiple instances on same device
- Peer-to-peer connection copying (when a client connects to a client that has already established connections, connecting client should establish those same connections)

+ Add a card

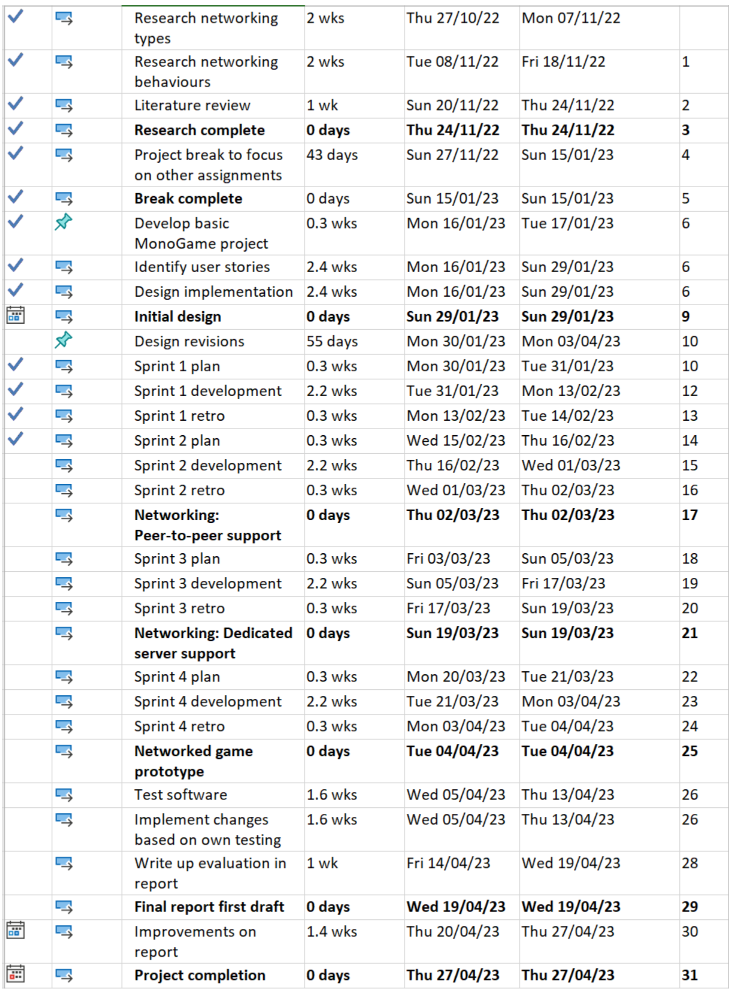
Figure 6.2: Sprint 2 task plan ("To Do": tasks chosen for this sprint. "Done": tasks completed in previous sprint(s))

Moving to sprint 2 then, the same user stories that were being worked on in sprint 1 were moved into the backlog. The goal of this sprint was to finish support for peer-to-peer, the developer said the following:

"I believe this has been done, at least at a prototype level. There is now support for NetworkedVariables and NetworkedGameObjects, with synchronisation packets being sent through all connections, and each client processing them correctly. Some work has been done on diagnostics, packet loss can be detected, although it still needs further testing."

Mention was made of the scrapped remote object creation, with the new solution being "a good enough compromise". During this sprint, a midway review took place. One of the main points made in this midway review was surrounding the lack of unit testing and how it would be a suitable fit for the project. This suggestion was taken on, and the project was then re-scoped to accommodate it.

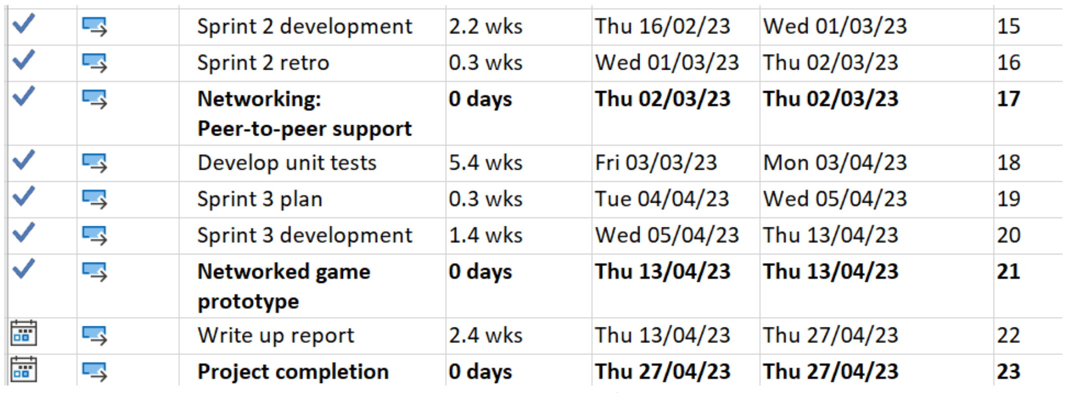
Unit tests would need to be written for all existing features as well as any additional planned features, it was going to take some time to do so. This is why it was important to reconsider what was desired to be completed in the project. Initially, the plan shown in figure 6.3 detailed multiple architectures being supported, firstly peer-to-peer. Followed by peer-to-peer with authority and client – server. Instead, the focus was shifted towards peer-to-peer with unit testing. So, sprints were halted whilst development of unit tests took place.



✓	⇨	Research networking types	2 wks	Thu 27/10/22	Mon 07/11/22	
✓	⇨	Research networking behaviours	2 wks	Tue 08/11/22	Fri 18/11/22	1
✓	⇨	Literature review	1 wk	Sun 20/11/22	Thu 24/11/22	2
✓	⇨	**Research complete**	**0 days**	**Thu 24/11/22**	**Thu 24/11/22**	**3**
✓	⇨	Project break to focus on other assignments	43 days	Sun 27/11/22	Sun 15/01/23	4
✓	⇨	**Break complete**	0 days	Sun 15/01/23	Sun 15/01/23	5
✓	📌	Develop basic MonoGame project	0.3 wks	Mon 16/01/23	Tue 17/01/23	6
✓	⇨	Identify user stories	2.4 wks	Mon 16/01/23	Sun 29/01/23	6
✓	⇨	Design implementation	2.4 wks	Mon 16/01/23	Sun 29/01/23	6
📅	⇨	**Initial design**	**0 days**	**Sun 29/01/23**	**Sun 29/01/23**	**9**
	📌	Design revisions	55 days	Mon 30/01/23	Mon 03/04/23	10
✓	⇨	Sprint 1 plan	0.3 wks	Mon 30/01/23	Tue 31/01/23	10
✓	⇨	Sprint 1 development	2.2 wks	Tue 31/01/23	Mon 13/02/23	12
✓	⇨	Sprint 1 retro	0.3 wks	Mon 13/02/23	Tue 14/02/23	13
✓	⇨	Sprint 2 plan	0.3 wks	Wed 15/02/23	Thu 16/02/23	14
	⇨	Sprint 2 development	2.2 wks	Thu 16/02/23	Wed 01/03/23	15
	⇨	Sprint 2 retro	0.3 wks	Wed 01/03/23	Thu 02/03/23	16
	⇨	**Networking: Peer-to-peer support**	**0 days**	**Thu 02/03/23**	**Thu 02/03/23**	**17**
	⇨	Sprint 3 plan	0.3 wks	Fri 03/03/23	Sun 05/03/23	18
	⇨	Sprint 3 development	2.2 wks	Sun 05/03/23	Fri 17/03/23	19
	⇨	Sprint 3 retro	0.3 wks	Fri 17/03/23	Sun 19/03/23	20
	⇨	**Networking: Dedicated server support**	**0 days**	**Sun 19/03/23**	**Sun 19/03/23**	**21**
	⇨	Sprint 4 plan	0.3 wks	Mon 20/03/23	Tue 21/03/23	22
	⇨	Sprint 4 development	2.2 wks	Tue 21/03/23	Mon 03/04/23	23
	⇨	Sprint 4 retro	0.3 wks	Mon 03/04/23	Tue 04/04/23	24
	⇨	**Networked game prototype**	**0 days**	**Tue 04/04/23**	**Tue 04/04/23**	**25**
	⇨	Test software	1.6 wks	Wed 05/04/23	Thu 13/04/23	26
	⇨	Implement changes based on own testing	1.6 wks	Wed 05/04/23	Thu 13/04/23	26
	⇨	Write up evaluation in report	1 wk	Fri 14/04/23	Wed 19/04/23	28
	⇨	**Final report first draft**	**0 days**	**Wed 19/04/23**	**Wed 19/04/23**	**29**
📅	⇨	Improvements on report	1.4 wks	Thu 20/04/23	Thu 27/04/23	30
📅	⇨	**Project completion**	**0 days**	**Thu 27/04/23**	**Thu 27/04/23**	**31**

Figure 6.3: Initial project plan, before midway review

Once unit tests had been written for the existing features, new unit tests would be worked on for the new features that had yet to be implemented. This included round trip time calculations, packet loss percentage calculations, client timeouts as well as the logger class. All of which were developed during the third sprint shown in figure 6.4.



✓		Sprint 2 development	2.2 wks	Thu 16/02/23	Wed 01/03/23	15
✓		Sprint 2 retro	0.3 wks	Wed 01/03/23	Thu 02/03/23	16
✓		**Networking: Peer-to-peer support**	**0 days**	**Thu 02/03/23**	**Thu 02/03/23**	**17**
✓		Develop unit tests	5.4 wks	Fri 03/03/23	Mon 03/04/23	18
✓		Sprint 3 plan	0.3 wks	Tue 04/04/23	Wed 05/04/23	19
✓		Sprint 3 development	1.4 wks	Wed 05/04/23	Thu 13/04/23	20
✓		**Networked game prototype**	**0 days**	**Thu 13/04/23**	**Thu 13/04/23**	**21**
📅		Write up report	2.4 wks	Thu 13/04/23	Thu 27/04/23	22
📅		**Project completion**	**0 days**	**Thu 27/04/23**	**Thu 27/04/23**	**23**

Figure 6.4: Project plan after unit tests

6.7 How could this have been done better?

Firstly, if unit testing was planned for from the beginning, it is safe to assume that the project would have gone a whole lot smoother. They were introduced late into the project and took a long time to be implemented which caused a big rescope. The main reason they took so long to implement was because of the impact it had on the developer. After making great progress with the second sprint, the introduction of unit testing seemed to slow the project down significantly, even if that wasn't necessarily the case. But this perceived lack of progress whilst working towards this new goal was enough to hinder motivation. The goal was still reached, and the newly planned scope was successfully developed, but if unit tests were planned from the start, this lack of motivation would have been a lot more manageable, if not avoided.

When it comes to the design of the software, one clear improvement that could be made is the introduction of threading, at least on a larger scale than it is currently used. As of now, threading is only used when a packet is acknowledged and the PacketSent method in the connection class. If threading was more involved in the whole design, then the performance could be improved significantly by spreading the workload of things like packet processing across multiple threads. For example, reflection is heavily relied upon for retrieving and updating networked variables which could be playing a role in the large performance drops, running these operations on separate threads could help. However, threading can be a difficult thing to manage, so doing so would take some time to make sure that data is being shared safely across multiple threads.

Another area of improvement is packet send rate, an area that reflection plays a huge role in. It would be very beneficial to slow down the rate at which packets are sent, even separating this from the game speed entirely. This would not only reduce the frequency of high-cost operations like reflection, but also reduce the possibility of thread contention, moments where threads are sitting idle waiting for a lock to be released (if the threading improvement was added as detailed previously).

7 Conclusion

The overall goal of the project was accomplished, and the agile methodology was followed throughout most of the development (apart from when the development halted). The library has been designed in a way that is very easy to pick up and use, allowing a developer to expand upon their own existing game with multiplayer functionality, albeit very performance heavy and barebones in feature support, which is one area of disappointment. The requirements stated the importance of ensuring that the performance decrease never surpassed 40% and unfortunately, that has been the case. However, the addition of unit testing has provided a great avenue of further work, with any further work able to be tested against the existing work. It could even be a great starting point for a developer who wants to learn more about networking but could potentially be overwhelmed by starting a multiplayer project from scratch, instead allowing them to experiment with an existing toolbox.

References

Awati, R. (2023) *What is ack (acknowledgement) in computer networking?* Available online: https://www.techtarget.com/searchnetworking/definition/ACK [Accessed: April 25, 2023].

Clement, J. (2023) *U.S. best selling games 2022.* Available online: https://www.statista.com/statistics/1285658/top-ranked-video-games-sales-annual/ [Accessed: April 22, 2023].

Fiedler, G. (2008) *Reliability and congestion avoidance over UDP.* Available online: https://gafferongames.com/post/reliability_ordering_and_congestion_avoidance_over_udp/ [Accessed: April 26, 2023].

Fiedler, G. (2008) *UDP vs. TCP.* Available online: https://gafferongames.com/post/udp_vs_tcp/ [Accessed: April 25, 2023].

Fiedler, G. (2010) *What every programmer needs to know about game networking.* Available online: https://gafferongames.com/post/what_every_programmer_needs_to_know_about_game_networking/ [Accessed: April 26, 2023].

Fortinet (2023) *What is TCP/IP in networking?* Available online: https://www.fortinet.com/resources/cyberglossary/tcp-ip [Accessed: April 25, 2023].

Khan Academy (2020) *Transmission control protocol (TCP).* Available online: https://www.khanacademy.org/computing/computers-and-internet/xcae6f4a7ff015e7d:the-internet/xcae6f4a7ff015e7d:transporting-packets/a/transmission-control-protocol--tcp [Accessed: April 25, 2023].

Knoop, J. (2019) *Bethesda Knew Fallout 76 'Would Have Bumps' – IGN Unfiltered.* Available online: https://www.ign.com/articles/2019/06/02/bethesda-knew-fallout-76-would-have-bumps-a-ign-unfiltered [Accessed: April 22, 2023].

Linus Tech Tips (2019) *Does High FPS make you a better gamer? Ft. Shroud - FINAL ANSWER* [Video]. Available online: https://youtu.be/OX31kZbAXsA [Accessed: April 26, 2023].

Microsoft (2021) *[MC-DPL8CS]: Host migration (peer-to-peer).* Available online: https://learn.microsoft.com/en-us/openspecs/windows_protocols/mc-dpl8cs/c188116b-228c-4c39-9959-381845f3d1af [Accessed: April 25, 2023].

Neumann, C. et al. (2007) "Challenges in peer-to-peer gaming," *ACM SIGCOMM Computer Communication Review,* 37(1), pp. 79–82. Available online: https://doi.org/10.1145/1198255.1198269.

Noland, K. (2019) "Mesh Networks" [Blog post] *Mesh Networks - Fundamental Games*, 27 June. Available online: https://fundamentalgames.com/2019/06/27/mesh-networks/ [Accessed: April 23, 2023].

Roxl, R. (2021) *What is peer-to-peer gaming, and how does it work?.* Available online: https://vgkami.com/what-is-peer-to-peer-gaming-and-how-does-it-work/ [Accessed: April 25, 2023].

Shapiro, R. (2020) *Myra/LICENSE.txt.* Available online: https://github.com/rds1983/Myra/blob/master/LICENSE.txt [Accessed: April 27, 2023].

Treese, T. (2018) *Evolve is being shut down but there will be one way to play still*. Available online: https://screenrant.com/evolve-servers-shut-down-2k-games/ [Accessed: April 23, 2023].

Velimirovic, A. (2021) *What's a dedicated server for gaming?.* Available online: https://phoenixnap.com/blog/what-is-a-dedicated-server-for-gaming [Accessed: April 23, 2023].

Wagner, B. (2023) *Attributes and reflection.* Available at: https://learn.microsoft.com/en-us/dotnet/csharp/advanced-topics/reflection-and-attributes/ [Accessed: April 23, 2023].

Webroot (2004) *What is P2P network sharing?* Available online: https://www.webroot.com/gb/en/resources/glossary/what-is-peer-to-peer-networking [Accessed: April 22, 2023].

Zou, D. et al. (2015) “A genetic algorithm for detecting significant floating-point inaccuracies,” *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering,* pp. 529-539. Available online: https://doi.org/10.1109/icse.2015.70.

Appendix A

Appendix A.1: User stories)

DEV 001: Matchmaking

As a: game developer

I want: to use a matchmaking system

So that: my game can allow players to join each other without needing a unique code, password, or IP address

----

Can a player search for an existing session?

Can a player join a session without needing to know the other player’s IP address?

Can a player join a session without needing to manually input some sort of “lobby code”?

DEV 002: Reusability

As a: game developer

I want: the library to be re-usable without needing any game specific code edits

So that: I can use it with multiple games with different implementations

----

Can the library be used in multiple, different projects without any library-side changes being made?

DEV 003: Diagnostics

As a: game developer

I want: to have access to network diagnostics data such as latency and packet loss

So that: I can display this information to players

Is diagnostic information available to be referenced outside of the library?

Is this information updated frequently enough to be considered relevant?

DEV 004: Peer-to-peer

As a: game developer

I want: to setup a peer-to-peer game

So that: my players can sync their locally running game session with other players who are also running the game locally

----

Can the library create a single instance of a peer-to-peer session?

Can the session store a list of connected players and their connections to each player in the session?

Can the library handle the sending and receiving of packets through all connections?

Can the session handle a player connection and disconnection?

DEV 005: Client / server

As a: game developer

I want: to setup a client / server game

So that: my players can join a host who is running the game code locally so that each player in the connection can contribute to and receive updates of the game state, without running the game code locally

----

Can the library create a single instance of a client/server session?

Can the host store a list of connected players?

Can each client store a connection to the host?

Can the library handle the sending of packets from host->all players and sending of packets from player->host?

Can the host handle a player connection and disconnection?

Can the session handle “host migration”?

DEV 006: Dedicated server

As a: game developer

I want: to setup a game on a dedicated server

So that: my players can directly connect to a dedicated server and sync their game with other players connected to that server without running game code locally

----

Can the library create multiple instances of a dedicated server session?

Can the server store a list of all connected clients?

Can each client store a connection to the server?

Can the server handle sending and receiving of packets to connected clients?

Can each client handle sending and receiving of packets to the server?

Can the server handle a player connection and disconnection?

Can each client handle server timeout?

DEV 007: Flow control

As a: game developer

I want: the library to handle flow control

So that: I don't accidentally flood a connection and induce severe latency

----

Can round trip time (RTT) be measured?

Can the flow control algorithm detect bad network conditions and change packet send rate accordingly?

DEV 008: Seamless connect / disconnect

As a: game developer

I want: to be able to handle connection and disconnection

So that: I can deliver a seamless experience to players in a session, that isn't limited to a fixed player count

----

Can I interface with the library to handle player connect and disconnect events?

Is a new player able to connect to an ongoing session without impacting the gameplay experience?

Is a player able to disconnect from on ongoing session without impacting the gameplay experience?

Can my game function without needing to fill a specific number of required players?

DEV 009: Client-side prediction

As a: game developer

I want: my game to support client-side prediction

So that: a client who isn't running game code locally, can still simulate their own player actions locally to provide a smoother gameplay experience

----

Does the library ensure the server holds authority by providing corrected simulation data in the event of a disagreement?

DEV 010: Reliability

As a: game developer

I want: the networking library to detect packet loss using acknowledgements

So that: network conditions can be tracked, and any important data can be resent

----

Do packet payloads include acknowledgements corresponding to the sequence number values of the current connection?

Are important packets stored so they can be resent if needed?

Is sequence number wrap-around automatically handled?

Can, at least, 32 sequence numbers be acknowledged in a single packet whilst still leaving enough space for the rest of the packet data?