Minimising Lag in Game Networking Models Without a Central Server

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

Szymon Jackiewicz
Student Number: 150249751
Supervisor: Dr Graham Morgan

Word Count: ???

MComp Computer Science w Games Engineering
Stage 3

Department of Computer Science
Newcastle University
18 April 2019

Declaration

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

Acknowledgements

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

Minimising Lag in Game Networking Models Without a Central Server

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

Szymon Jackiewicz

Abstract

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

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

Contents

Chapter 1: Introduction

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

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

1.1 Project Goals

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

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

Implementation goals:

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

Data gathering goals:

- TODO

1.2 Basic concepts and terminolory

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

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

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

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

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

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

1.3 Networking principles in games

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

1.3.1 Communitaction Issues

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

Data Distribution

Broadcast to each client

Latency

Lag and Jitter

Reliability

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

Bandwidth

Consumer hardware and networking slow (esp upload)

1.4 Networking Model Options

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

1.4.1 The Centralised Server Model

Most AAA online multiplayer games that are played today, make use of the "centralised server model" for synchronising the simulation state between several clients participating in the same simulation. This means that in an example of a First Person Shooter (FPS), if one player presses the "jump" key, their character will jump and this information is would also be sent over to the game server. The server would then send the information that this player has jumped, to all other clients. There is a potential problem here however. Given that the ping between the server and client A is α and between the server and client B is β, the time between client A pressing an input and client B being notified of this input can not be less than $\alpha + \beta$ and due to the limitations of physics $\alpha > 0$ and $\beta > 0$. This means that at any given time the lag experienced between clients A and B will be more than $\alpha + \beta$ when processing times are factored in too. This could lead to a problem of a poor experience for a client with a high ping to the server, as they will receive the updates from the server later than every other player and therefore be at a disadvantage if the game requires real time reactions. Unfortunately under some implementations, this also results in a poor experience for every other player, who despite having resonable ping to the server, can be shot from behind cover by a laggy player who fired a shot before the cover was reached on their version of the game state which is delayed compare to others.

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

Possible solutions to the variable ping problem

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

1.4.2 Client Hosted Model

The client hosted model is an example of implementing a networking infrastructure without the need for expensive server rental. One of the main advantages of this solution

- ✔ Hardware is likely to be powerful enough to handle the stress of many simulations running simultaneously.
- ✔ High bandwidth connection is likely to be used. Minimises the chance of high latency and network issues.
- ✔ Ping to the server is likely to be similar for all players making the game more fair.
- ✔ No client can see the address of any other client.
- ✔ The developer has a lot of control over what is allowed within the game. This allows for anti-cheating systems that are hard to bypass.
- ✘ Expensive to rent out or buy and maintain server space. Letting players rent out servers makes the game more expensive for them.
- ✘ Many servers have to be spread out evenly throughout the world to allow for low latency connections.
- ✘ People living in remote locations may not have low latency access to official servers.

Figure 1.1: The attributes of the central server model

is the idea that one codebase can be written to work as a client hosted model and this codebase can then be adjusted slightly to work on a central server as well if there ever is a need for this in the future. The basic idea behind this implementation method, is that one of the clients, would act as a server for all other participants alongside also being one of the participants.

- ✔ Test good
- ⊖ Test attribute
- ✘ Test bad

Figure 1.2: The attributes of the client hosted model

1.4.3 Peer to Peer Model

TODO

- ✔ Test good
- ⊖ Test attribute
- ✘ Test bad

Figure 1.3: The attributes of the peer to peer model

Chapter 2: Research

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

2.1 Networking implementations in games

TODO:

2.1.1 Variable ping system in Battelfield 1

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

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

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

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

2.1.2 Networking in Apex Legends

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

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

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

2.2 Other networking developement tools

TODO:

2.2.1 Unity's UNET

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

2.2.2 Photon Network in Unity

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

2.2.3 Google's Stadia

Allows for easy networking since it's already on server

Chapter 3: Design

3.1 Code Architecture

Test for inline: `test Hello;`

3.2 Protocols

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

3.2.1 Protocols in Network Communications

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

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

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

could be sent at that time instead. Simply, old information is quickly outdated and it's more important to send new information then old, non-useful information.

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

3.2.2 Connecting the clients together

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

Design of the matchmaking server

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

1. Initialise Networking Library with predefined, known port
2. Listen for "Join Request" messages on the port from clients.
3. If the client is allowed to join the game, reply with "Join Acknowledgement" message.
4. (a) If the client is a P2P session host
   Broadcast the information about each client to each client.
   (b) If the client is a CH session host
   Send the information about each client to the game server.

I have experimented with different approaches for implementing this flow. Initially, for simplicity, the implementation consisted only of UDP messages being sent between clients.

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

3.2.3 Main Game loop and Broadcasting

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

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

out of scope of this project, the way the packets are sent from a server to the clients, can make a process like this easier.

The most common implementation for how servers broadcast data from the server to clients in AAA games, is to broadcast all necessary update data at a constant rate many times a second. If a packet is not delivered to one of the clients, there is no need to resend it since a new updated packet will be sent right after the last one. The constant rate at which the packets are sent and the idea that the same amout of time in-game time has passed between each consecutive packet that is received, makes startegies such as dead reckoning easier to implement than is the updates were sent irregularily.

Update rates

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

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

Tick rates and simulation steps

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

The term tick rate refers to how many times per second, the game server will process and produce data. At a tick rate of 60Hz, the time between any two ticks would be 16.66ms meaning that the server will have a processing window of that much time to process and broadcast a simulation step. If the server manages to finish the processing

of a simulation step within this window (i.e. in this example, processing took less than 16.66ms to complete), it would sleep until the next simulation step needs to be processed. A short processing time on each simulation step means that the clients receive their updates earlier which will lead to reducing the ping between players and make systems like hit registration feel more responsive in gameplay. When deciding on what tick rate is best to use for the server, it isn't always correct to set the tick rate high as possible. It is also very important to consider that in the most likely worst case scenarios the processing of each simulation step is shorter than the time between ticks. It is paramount that this processing is done within the time window as the server processing not keeping up with the tick rate, often results in inconsistencies such as failing physics or clients rubber banding therefore proving a suboptimal experiance for the clients.

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

3.2.4 Message Structure Strategies

Data Representation

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

There are many different ways that the same data can be represented and each one is carefully designed to be the most appropriate for it's use. Consider the two figures below of common ways of grouping complex data in an "easily readable" format; XML in Figure 3.1 and JSON in Figure 3.2. These example have been adapted from the article *Reading and Writing Packets*(Fiedler).

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

Figure 3.1: An example of a representation of world data in the XML format

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

Figure 3.2: An example of a representation of world data in the JSON format

Detecting and Dealing with Packet Loss

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

TODO: Find source where I read this... each update packet can also contain a copy of the previous packet attached to it so that even if packet loss occurs, the simulation can stay up to date.

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

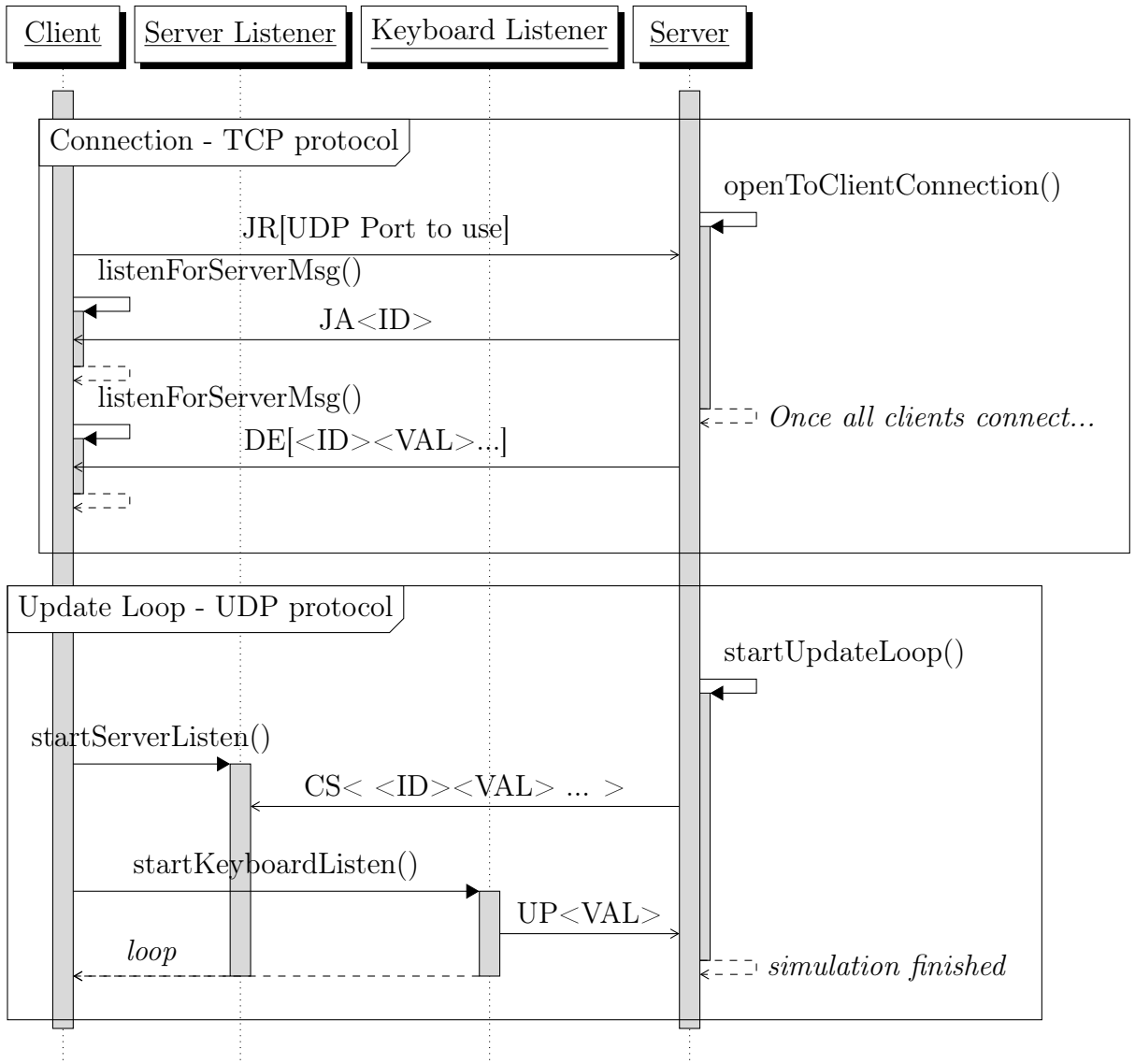

Figure 3.3: Graph showing the protocol of joining a session hosted on the server as well as sending and receiving updates.

3.2.5 Potential issues with the Client Hosted protocol

The protocol for establishing a connection and transfering of data can be found in Figure 3.3. There are many potential flaws with this approach.

Packet Loss

Time outs in place in case response lost... resend.... Resend request if no response....

Security

anyone could send update by spoofing ip

3.3 Message Codes

Message Type	Message Code	Description	Example Payload
Join Request	`JR`	Allows a client to send a join request to the server.	`JR`
Join Acknowledgement	`JA`	Allows the server to confirm that the client's information has been saved. Is followed by 1 byte indicating the client's ID	`JA1`
Ping Request	`PQ`	Message instructing the recipiant to reply with `PS`. Can be used to time the delay in this connection.	`PQ`
Ping Response	`RS`	This should be sent whenever a `PQ` message is received.	`RS`
Update	`UP`	Used by a client to update it's value on the server. Is followed by 1 byte representing the new value.	`UP9`
Define	`DF`	Used by the server to define the initial values for each of the clients connected to this instance. It is followed by a non-zero, even amount of bytes representing the client ID and it's value pair.	`DF1020304050`
Current State	`CS`	Used by the server to broadcast it's real state to all clients. When this is received, clients are expected to update their local state to this. It is followed by a non-zero, even amount of bytes representing the client ID and it's value pair.	`CS1927344157`

Table 3.1: Table showing the message codes for distinguishing messages from each other and how each one is to be used

Appendix A: Appendix Title

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

References

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

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

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

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

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

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