Adding Networking to a First Person Shooter Research Game

A Major Qualifying Project (MQP) Report
Submitted to the Faculty of
WORCESTER POLYTECHNIC INSTITUTE
in partial fulfillment of the requirements
for the Degree of Bachelor of Science in

Computer Science

By:
Alexander Hayden
Benjamin Peters

Project Advisor:
Professor Mark Claypool

Sponsored By:
NVIDIA

Date: December 2022

Abstract

First Person Science, or FPSci, is a high-performance, light-weight first person shooter developed by NVIDIA to facilitate experiments involving latency and human-computer interaction. Unfortunately, FPSci only supports single player experiences. To enable the simulation of a wider range of scenarios typical of first person shooters, we designed and implemented a multiplayer mode. Our version uses a client-authoritative model, utilizing two UDP connections between each host and the server to achieve fast and reliable delivery of updates and control messages. Our code is effective, supporting more than 64 simultaneous clients and a 400Hz refresh rate.

Acknowledgements

This project would not have been possible without the encouraging support of many people. We would like to thank the FPSci team at NVIDIA, Ben Boudaoud, Josef Spjut, and Joohwan Kim, for developing FPSci, sharing their knowledge, and guiding us as we learned about and worked on the project. We would also like to thank Professor Mark Claypool of the WPI Computer Science Department for advising, guiding, and supporting us throughout the project. Finally, we would like to thank Samin Shahriar Tokey, Ivan Klevanski, Sitsanok Young, Yihong Xu, and Alex Mitchell, fellow WPI students also working on FPSci, for sharing ideas, code, and support as we worked.

Contents

List of Figures

Listings

Chapter 1

Introduction

Video games are one of the most popular forms of entertainment consumed today. With fast and high-capacity Internet connections readily available, many modern games feature network components: global scoreboards, pushed updates, and especially networked multiplayer. With the growing popularity of games, it is important for game makers to produce high-quality products, and to do so they must understand how players and games interact. This understanding is also applicable to general human computer interaction and is useful outside the game industry.

First Person Science (shortened to FPSci) is a first person shooter game developed by NVIDIA with a focus on high performance and easy configurability. Although it offers considerable control over a single user experience, FPSci currently has no support for studies with multiple players in the same match. There is no network code at all in the original FPSci implementation, which prevents its use in studies that involve interactions between players.

We designed and implemented a networked multiplayer mode for FPSci, allowing multiple players to connect to a server and play together. We developed these features with the goals of changing as little functionality as possible, keeping performance high, and being configurable to support experiments.

To achieve this, we first determined a network structure, utilizing a client/server model and UDP communications. We selected the networking library ENet to build on top of and decided to use both the reliable communication channel provided by ENet and an unreliable bare socket for each connection. We defined protocols for how data should be encoded, sent over the network, and decoded. We then implemented these protocols, adding code for serializing and deserializing data to interface the communicated information with the FPSci game world. This involved writing code in the game loop to interact with game objects, code to interface with ENet, and layers in between to consolidate

boilerplate code, standardize our data structures, and make the code readable.

Once we determined that our code worked, we ran experiments to measure how it performs under a variety of conditions. We examined variables such as framerate, bitrates over the network, CPU utilization of our code, the latency between clients and the server, and the world state deviation between clients. We varied the framerate of the game, the number of clients connected in a single match, the latency of the network connections, and the packet loss rate of the network.

We found that our work yielded acceptable performance. Our version of FPSci was able to run multiplayer matches at upwards of 400Hz, handle more than 64 simultaneously connected clients, and continue to run at pings up to 200ms (albeit with noticeable differences between the client's world states). We found that almost all resource usage scales linearly with increasing framerate. The limiting factor for large matches is the exponential growth of network usage on the server as more clients join.

Below, we outline the process we took in creating this new version of FPSci. Chapter 2 offers a a background overview of FPSci, its dependencies, and networked multiplayer in games - including the challenges it presents to would-be-implementers. Chapter 3 goes into depth on our implementation - how we approached development, our choices of tools, the network communication scheme we developed, and the classes and functions we wrote. Chapter 4 covers our evaluation, including the methods we used to measure performance, how we ran our experiments, and details from our results. Chapter 5 recaps our work, summarizes our design requirements, discusses the results of our evaluation, and examines potential future work on this project.

Chapter 2

Background

It is important to understand the motivation behind and structure of FPSci before embarking on any effort to expand it. FPSci allows researchers to change parameters of interest using a hierarchical plug-and-play configuration structure. It also features per-frame logging to make in-depth analysis easier. The game is built on the G3D Innovation Engine which abstracts away many of the lower level data structures and object handling, and provides garbage collection. To create features involving network connections, it is useful to understand established approaches to networked multiplayer and latency compensation techniques to inform implementation decisions during development.

2.1 FPSci

FPSci is a simple first-person shooter game used for performing experiments. The project was started by a team at NVIDIA investigating how latency affects players' performance and behavior in first-person shooters. As of this writing, FPSci consists of a first-person shooter interface, a variety of maps, and several types of targets to shoot[1]. The platform is lightweight and capable of running at over 400Hz on modern hardware. This high tick rate allows for precise control over latency - there are configurable parameters to set the latency between the user and the interface and within the game's data pipeline. There are also configurations available for the targets the player interacts with and common attributes of the player. In order to make running experiments easier, FPSci also features a modular plug-and-play format for specifying experiment parameters and robust logging to a SQLite database to make the analysis of results easier.

FPSci enables study of latency in first person shooters. Latency, or "lag," has a definite impact on how the game runs and how the player interfaces with it, in turn affecting the player's enjoyment of the game. Studying the ways

that player performance is impacted by different parameters is an important step in improving the player experience in first person shooters and gaining a deeper understanding of human-computer interaction. Studying these effects in triple-A games provides the benefit of a realistic study environment but does not provide the ability to carefully control all aspects of the experiment, possibly causing results to be confounded by variables and constraints outside the researchers' control and making it difficult to replicate an experiment to verify results [1]. To address this, FPSci was built as a high-performance application for user studies investigating the relative impacts of latency and framerate on users and has since been expanded to support user studies with broader control requirements [1]. FPSci allows a researcher to specify many parameters for experiments including framerate, frame delay, target parameters, feedback questions, logging controls, player movement, world physics, commands to be run based on experiment progression, and more [2].

FPSci has a simple and flexible format for specifying parameters and controlling the flow of experiments. It follows a three-tier approach to experimental design, consisting of experiments, which contain one or more sessions, which in turn contain one or more trials. The experiment level contains two types of settings: universal settings and general settings. Universal settings are settings that cannot be changed during an experiment, including target configuration and some experiment flow settings. General settings can be overridden at the session level, effectively making them default values when defined at the experiment level. General settings include rendering settings, weapon specifications, player configurations, feedback configurations, logging settings, and experiment flow settings. The experiment level is where the majority of general settings are defined because they do not change throughout the course of an experiment. However, any of the general settings can be changed at the session level in order to study their effect. Sessions are the lowest level where settings can be changed. Usually, an experiment's independent variables are manipulated between sessions. Each session contains a series of trials. Each trial consists of a list of targets to spawn and a number of times that trial should be repeated.

FPSci makes use of G3D's Any file specification to store configurations. Any is a JSON-like format that allows for simple parameters and more complex objects to be described in plain text. The Any specification also supports nesting data objects and custom parsing, allowing for modular configuration files that are easy to read, modify, and maintain. By default, FPSci makes use of 6 Any files to describe an experiment. The purpose, and structure of each file is detailed below:

- experimentconfig.Any: Contains 4 major subsections
  - Experiment-wide settings- These are settings that are consistent for the whole experiment, but can be overridden by session level
  - Sessions

        * Each session can override any of the experiment-level settings
        * Trials- Specifies the number of times that a session should be repeated (no settings changed)
    - Targets- Indicates how targets should be drawn and move
    - Weapon- Indicates how the weapon should work

- keymap.Any- Maps player actions to keys/buttons
- startupconfig.Any- Sets FPSci searches for other Any files at launch
- systemconfig.Any- Settings for latency logger devices
- userconfig.Any
    - Session/Experiment agnostic user settings
        * Mouse DPI/sensitivity
        * Reticle preferences
- userstatus.Any- Defines which sessions a user should do and in what order

While running, FPSci keeps a detailed log of what happens in the game. This allows researchers to analyze player behavior and performance after running an experiment. FPSci logs are stored in a SQLite database that is managed by a dedicated thread. In the interest of performance, the contents of the logs are not written out to disk until either the memory fills up or the experiment ends. The database includes tables that hold values for each experiment, such as the experiment config, as well as per-frame information on frame timings, physics, and player activities such as aiming and shooting. Feedback questions and their responses, session parameters and statuses, target parameters, positions, and trajectories, trial performance details, and user settings are also logged in the database. Additionally, if the logs do not contain the session parameters of interest, other parameters can be added to the logs by modifying the experiment config. By logging relevant values on a every frame, FPSci stores information on what happens the entire time a trial runs. Furthermore, by organizing these values into a SQL database, the logs remain easy to parse and the table structure allows researchers to easily access and format all the data they need by querying the relevant tables.

2.2 Structure of G3D

FPSci is built using the G3D Innovation Engine [3] - an open-source game engine written in C++. In addition to data structures such as strings and arrays, and services like garbage collection, G3D also grants access to the structure of inheritance that defines how a game runs. This allows developers to modify and change almost everything G3D does for their game if they desire. All

G3D apps must define a class that extends the G3D GApp class, which contains default functions that handle all the activities that happen within the game: graphics, AI logic, networking, physics simulation, user input, and more. All of the activities that happen in each frame are invoked within a method called oneFrame(). By overriding the oneFrame() method, a developer can define a custom order for the events that happen within the game loop, or a fully custom set of events. By overriding the methods called in oneFrame(), a developer can add custom behavior to their game. Most of the code modifications we implemented during this project were in the onNetwork() method, which is called once with each invocation of oneFrame(). G3D also provides parsing, writing, and other low-level functionality for using Any files. This simplifies the process to add additional parameters to FPSci by taking advantage of systems already defined by G3D to parse files and generate complex object types.

2.3 Traditional approaches to networked multiplayer games

Split-screen and multiple-controller multiplayer games have been around since the earliest days of video games, but many modern games utilize the Internet to allow players to play together on separate hardware. Although the modern Internet can usually deliver data quickly, data delivery over the Internet is inconsistent: some packets may be lost and round-trip times may vary from packet to packet. These properties of Internet traffic necessitate engineering to ensure that the user experience is responsive, accurate, and consistent between players.

At its simplest, network code in games (or "netcode" as the industry calls it) is responsible for synchronizing a shared game between players. Barring more advanced style games, networked play functions the same as split-screen multiplayer: all players share the same world, but are able to interact with said world independently. In split-screen play, the game can load one single copy of the world, insert a camera for each player, and then as it receives inputs from each player, update the world. Each player sees the current world state because they are observing the same copy of the world, just from different viewpoints. In a networked game, every player's machine is responsible for rendering a copy of the world for that player to interact with. Generally, this means that the client (the copy of the game running on the player's machine) has a complete copy of the game world. The challenge then becomes ensuring that the copy of the world that each player has is consistent with the copy each other player has. For example: if one player sees a door open, and another sees the door closed, the consistency is broken: one player may get shot through what they believe to be a closed door, or the other player may be confused as to why they cannot shoot through an unobstructed doorway.

The exact methods for keeping clients consistent vary from game to

game, but the basic principle is whenever the game world of one player changes, the change must be communicated to all other clients as quickly as possible and replicated there. Writing good netcode requires consideration of the following [4]:

- Latency: Communication over the Internet is delayed. If, say, the connection from one client to another had a round-trip time of 40ms, updates made on one client would appear on the other 20ms later at the earliest. An object moving at running speed can move 6 inches in that time, which could be the difference between a hit and a miss in a first person shooter. This can also lead to clients disagreeing on the game state: one client may change something after another client has already changed it, but before it has been informed of the other client's change. This desynchronizes the game worlds and requires intervention to resynchronize the clients.
- Bandwidth: Although packaging and sending the entire game state to every client each tick could ensure they are consistent, that much data would congest the network, adding delay and degrading gameplay quality. Games running at modern frame rates (60 or 120fps) can easily send 120 updates per second. The data sent can add up quickly if each update contains a significant amount of data. This requires the writers of netcode to weigh the costs of sending data: only necessary data can be sent, and even then in moderation. Compressing data can help, but one should consider how this affects the accuracy of the received information and the added latency of compressing and decompressing the data. If not all data can be sent in a single tick, a system must be in place to prioritize some traffic over others and then deal with receiving part of the information late.
- Data errors: The Internet is not reliable, so designers cannot assume that data will arrive at the destination exactly the way it was sent. The netcode should be robust enough to handle missing, corrupted, or out-of-order packets without significantly disrupting the game.
- User Experience: Disruptions to the network, and even just latency, are noticeable to the players. Entities lagging behind their actual positions, shots that look like they hit but actually do not, and large jumps in the game state can confuse and frustrate players. A large part of netcode is working to hide - or at least minimize the effects of - these discrepancies for players. Although our project does not attempt any of these techniques, there is considerable research into "compensating" for latency, controlling bandwidth usage, and resolving conflicts between clients.

There are two main architectures used in networked games [5]: client authoritative and server authoritative. In a client authoritative model, clients either connect directly to each other, or the "server" acts only to forward messages between clients. If a client makes a change, it is assumed to be accurate

and replicated across the other clients. If there is a conflict, the clients communicate directly to determine which state is valid. In a server authoritative model, all clients communicate with a central server that also has knowledge of the game state. It accepts inputs from the clients, acts on them locally, and then sends the results to all clients. This means the server is in charge of resolving conflicts with an authoritative copy of the game world for reference, and it prevents clients from violating any game rules (either intentionally or unintentionally). Both approaches have merits and can be made to look indistinguishable from each other to a player. However, a developer's choice of approach has a huge impact on the design and performance of the game.

2.4 Latency and FPS Games

Spjut et al.[6] demonstrate that being able to separately control the amount of input latency and refresh rate of a game can reveal nuances in the relative impact of these distinct types of delay. Additionally, Lee et al. [7] show the impact of different server tick rates on player accuracy, highlighting the need for a high tick rate server for an evaluation platform. Liu et al. [8] find that even small reductions in local latency drastically increased player accuracy and score, indicating the importance of a lightweight test-bed when experimenting with latency. Henderson [9] finds that players appear to be willing to accept extremely poor network conditions and appear to care more about the difference in latency between clients than they care about the amount of latency between them and the server. Spjut et al. [10] explain that having accurate control over end-to-end latency can allow researchers to better understand the foundations of esports performance. Li et al. [11] describe a latency compensation mechanism for equalizing network latency for all players in a cloud gaming environment to study the impact of latency on Quality of Experience. Vlahovic et al. [12] show that VR games may be just as sensitive to latency as non-VR FPS games, further elevating the importance of understanding the principles of FPS games and players' behavior. Our work develops a tool that can be used to conduct similar research to further understanding in these areas.

Chapter 3

Implementation

After establishing our design goals and gaining an understanding of the original FPSci, we began designing and creating our additions. We started by making decisions: choosing a networking library, server design, and communication protocol. After this, we planned and implemented individual parts of the complete modification, and using an iterative development style to create a final product of quality, effective code.

3.1 The Development Process

Throughout the development process, we collaborated with other teams modifying and using FPSci. In addition to our project, there were three FPSci research teams at WPI. One was working on adding an authoritative server and latency compensation to networked FPSci for their Major Qualifying Project (MQP). The next was a master's student focusing on supporting any user studies run with the project and adding features necessary for those projects. Finally, there was a group that ran user studies using the networked version of FPSci.

Because all of these groups planned to work with the same code, we decided to use an iterative development process. To add a new feature we would start by determining the required characteristics of the feature and brainstorming a high-level approach to adding the feature. Once we knew the requirements and general approach, we would work to create a minimum viable product that met all of the requirements. With this done, we would review how it was actually implemented and make decisions about what changes would make the feature more robust and generalizable. Once we had decided what changes we wanted to make to the feature we would implement those changes and refactor the code we wrote in the first pass to be more understandable and maintainable. At this step, we would also add parameters to the appropriate configuration files so that researchers could specify different behaviors of the new feature without

having to modify the code. This allowed us to develop in parallel with the other groups: if other groups were dependent on a feature and planned to use the features soon, we would release the feature either as the minimum viable product or after only an initial round of refactoring, with the understanding that we would refactor and finish it and replace the released version eventually. In order to avoid conflicting changes, we tried to only work on features that were orthogonal to the work the other groups were doing, but we were not always successful.

3.2 Requirements

There were a few fundamental requirements we identified early in our process. The first and most important requirement we identified was that any changes we made would have to maintain backward compatibility with experiments run with older versions of FPSci. This is in line with the development requirements of FPSci where any researcher can use the latest version of FPSci with the experiment config file from their experiment, regardless of what version of FPSci it was written for. It was important that this remained true for any changes that we made so that we did not break any experiments designed before our changes were made. The second requirement that we identified was that our implementation should be a server/client design instead of a peer-to-peer design so that all players experience the same network conditions. Additionally, our implementation should maintain the general flow of experiments that FPSci currently supports with as few changes as possible. Finally, it is important that game and experiment parameters are configurable from files without having to make code edits, in line with how the original FPSci operates.

3.3 Decisions

Although we had a list of design requirements, it was up to us decide the specifics of implementation. This included deciding on the server structure, underlying networking library, and network structure for game data.

3.3.1 Server Structure

One of the first decisions we had to make was how to build the server application. We were first faced with determining how to handle client action resolution. Specifically, we had to decide if we wanted to build a server-authoritative system, where clients send their requested action to the server and then the server responds with the updated world-state after that action is applied, or a simpler client-authoritative system, where clients simulate all of their actions according to their local world-state and then send their updated

world-state to the server to be replicated to other clients. We considered several factors when making this decision:

The server could be significantly simpler if we went with a client-authoritative system because the server would simply have to store the world state, apply updates from the clients each tick, and forward that saved state to each of the clients so they can update any remote objects in their local world-state. Additionally, the client authoritative model can help clients hide latency because their actions are reflected locally immediately without waiting for the server to validate their actions. This is essentially the same as the latency compensation mechanism of *local prediction* which is sometimes implemented on systems where a server authoritative model is used.

However, there are significant downsides to client authoritative system. The most impactful of these is that because hit resolution takes place on the client of the shooter. As long as the client shoots where the target is on their screen, it registers as a hit *regardless of whether or not the other clients agree that object is in that location at that time.* Because of this, a server authoritative model is more accurate because it waits until it has all of the clients' positions for a specific frame before doing hit validation. However, this usually means that in order to hit a target in a server-authoritative system, a client has to "lead" (aim ahead of) the target by a variable amount that depends on the latency of the network. Additionally, most first-person shooter games make use of a server-authoritative model because this makes it significantly harder to cheat - the server does not treat updates from the client as truth, performing validation on updates which ensures that clients can not do illegal actions like flying or shooting a target without being able to see it.

After weighing these options carefully, we decided to use a client-authoritative model for our first iteration and then later add a server-authoritative option to the game if we had time. By building both of these systems, we could allow researchers to choose which model they want. This also allowed us to start with a simple server that does not need much logic, giving us more time to spend building network infrastructure and get familiar with the code base.

3.3.2 Network Library

Because of the inherent complexities of network communications, we utilized an external networking library in our FPSci fork rather than writing our own code from scratch. We examined eight different networking libraries before settling on a single one, ENet [13], to use in the project. Our search focused on network libraries geared toward games but did include more generic libraries. Based on the goals of our network implementation, low latency being the most important, we identified four criteria we wanted to be included in the library we chose:

1. UDP support: our implementation requires a low baseline latency which

may not always happen for TCP connections.

2. Ease of use: the library should handle the intricacies of network protocols, allowing us to focus on the application level of the network implementation.

3. Simple dependency: because all future FPSci developers need to set up the library, it should be easy to install and small in size.

4. Reliable communication: As well as standard UDP communication, we would like a way to send occasional packets over UDP with guaranteed delivery.

The libraries we examined are listed below:

ENet

ENet[1] is a simple and straightforward game networking library. Built on top of Berkeley sockets, its main feature is low-latency, reliable communications. It also provides simplified wrapper functions around the basic Berkeley socket functionality to aid in the use of raw UDP or TCP sockets. Reliable UDP communications offer control over retransmission controls and throughput while still being considerably more performant than TCP connections. The library is open-source and has been actively maintained for over a decade. It is also a dependency for G3D and is thus already available to FPSci developers.

GameNetworkingSockets

GameNetworkingSockets[2] is a modern game networking library built and maintained by Valve. It provides numerous features on top of UDP to ensure reliable communication, and includes methods for segmenting and prioritzing traffic. It also includes methods for simulating latency and packet loss and does logging. Although a promising choice in terms of features, it is a large added dependency and has a somewhat complicated build process.

Boost

Boost[3] (specifically the ASIO package) is an industry-standard networking and IO library. It offers complete functionality and control of UDP and TCP connections, but is slightly more complicated as a result. Because it is not designed for games it does not seem to offer reliable UDP functionality out of the box. It is also a large (although well-documented) dependency.

[1] `http://enet.bespin.org/index.html`
[2] `https://github.com/ValveSoftware/GameNetworkingSockets`
[3] `https://www.boost.org/doc/libs/1_79_0/doc/html/boost_asio.html`

Netlibrary

Netlibrary[4] is a smaller networking project designed for games. It offers four different levels of reliability applied to UDP connections, varying from reliable and ordered messages to unreliable and unsequenced messages. Netlibrary is lightweight and very simple to use, however, its methods are blocking by default, necessitating further work on our part to use them within the project.

SLikeNet

SLikeNet[5] is a fully featured game networking suite, built as a successor to the RakNet library. It offers rich functionality and is actively being developed. The extensive functionality means the library has a steeper learning curve as well as a much larger memory footprint. We decided it was overkill and not worth the time to learn for this project.

Yojimbo

Yojimbo[6] is a fully featured game networking suite designed especially for competitive first-person shooter games. It offers relatively straightforward functions and should be easy to use. However, it has special requirements for the game loop and may not easily allow flexibility for us to implement more novel latency compensation techniques.

Winsock

Winsock[7] is the standard windows library for socket operations included in the Windows SDK. It offers full use of TCP and UDP sockets but requires considerable infrastructure to be built on top of them. Many other networking libraries listed here are built on top of Winsock.

Other Libraries

We came across the following libraries in our research, but rejected them upon cursory inspection. They are included here for reference.

EVPP

EVPP[8] is a modern, enterprise-backed C++ networking library built

[4] https://github.com/matt-attack/netlibrary
[5] https://github.com/SLikeSoft/SLikeNet
[6] https://github.com/networkprotocol/yojimbo
[7] https://docs.microsoft.com/en-us/windows/win32/winsock/getting-started-with-winsock
[8] https://github.com/Qihoo360/evpp

with an extreme focus on performance. However, it requires numerous dependencies, is developed by a Chinese company, and most of the documentation in Mandarin.

SPAS (Small, Portable, Asynchronous Sockets)

SPAS[9] is a simple socket interface based on Boost/ASIO, implemented entirely in a single header file. SPAS is genuinely impressive and is certainly not a large dependency, but it, unfortunately, does not support UDP communications.

3.3.3 Network Structure

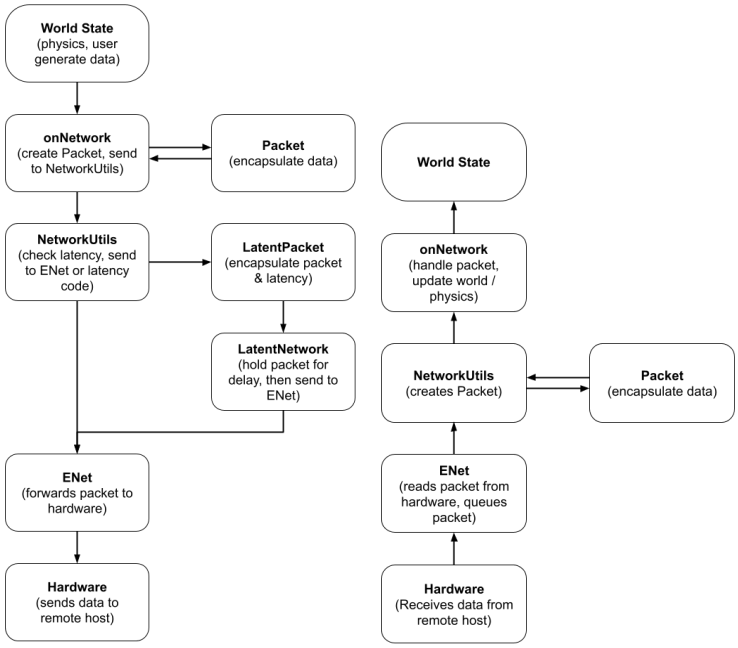
Before we could start implementing any of our changes, we also had to decide how we wanted to structure the network messages and the code for sending and receiving them. We needed to send some packets as fast as possible and some packets as reliably as possible, depending on the type. This would have been extremely difficult to implement on a single socket connection, because there would effectively be two streams of traffic on the same channel: one reliable and one unreliable. To solve this, we decided to open two connections between each client and the server. One, for reliable traffic, makes use of ENet's Peer structure, which provides reliable communication over UDP and an easy interface for sending and receiving packets. The second, for unreliable (but low-latency) traffic, is a standard UDP socket that we controlled using the ENet socket functions, which provide a basic wrapper over the Winsock library.

Our packet structure and construction code is covered in detail in section 3.4.3 . From the beginning, we knew that we wanted all the code responsible for creating and sending packets to be separate from the code that was responsible for interfacing with the game world. This led us to create a new class that we called NetworkUtils, which is responsible for both taking the game data, converting it to bit strings and then sending those over the network, and for converting the input from the network back int usable game data. While this was our plan from the beginning, we initially struggled to fully separate the game logic from the networking code and originally had significant game logic within NetworkUtils methods. We later remedied this through refactoring.

We also chose to implement a way to configure the latency of the links between client and server from within FPSci. Although a configured router on the network can easily add configurable latency, packet loss, and other variations to the traffic between clients and the server, it requires adding hardware to the test environment. FPSci is designed with experiments in mind; there are configuration options for almost all variables that could be experimented with. Thus, it makes sense to include network latency within the game itself, since most experiments involving networked gameplay will likely involve setting or adjusting the latency of clients. We inserted this functionality just above the

[9] `https://github.com/ruifig/czspas`

Figure 3.1: Path of outbound data (left) and inbound data (right) through networked FPSci



ENet layer, via our LatentNetwork class.

3.4 Implementation Details

3.4.1 Data Flow

Figure 3.1 shows broadly how information is passed through our code. Outbound data is generated by the simulation or player inputs. Our onNetwork function pulls these changes, encapsulates them in suitable packets, and calls NetworkUtils functions to process the packets. NetworkUtils checks the packets for connection information, then calls the correct boilerplate code to pass the packet contents to ENet, which in turn sends them to the OS and hardware. Inbound data is received off the hardware by the OS and queued in ENet. When onNetwork looks for incoming traffic, it triggers NetworkUtils to pull packets from the ENet queue, encapsulate them in our Packet class, then pass them back to onNetwork. onNetwork then updates the world state accordingly.

3.4.2 Changes to the onNetwork Loop

Most of the changes that we made to the operation of the FPSci app came from our modification of the onNetwork method. This method is called by the oneFrame method of the application to handle all network tasks once each frame. The actions that onNetwork performs differ between the server and client, but the general behavior is similar: The networking code starts by receiving and handling all incoming packets. These packets contain updates for networked entities, game state information, player activities such as shooting, and control traffic such as connection notifications and registration information. After all of these packets have been received and processed, and any updates have been applied to the current game state, an update of local changes is sent to the relevant remote party so they can update their game state. We implemented two onNetwork functions, one on the server and one on the client.

Server-Side onNetwork

The server-side loop acts as described above, with most of it's code dedicated to handling control messages about client status and activities, and only a small portion handling sending and receiving updates from clients. The function does the following:

1. Calls the parent's onNetwork function. This does nothing as far as we can tell, but is good practice in case GApp default network functionality is added.
2. Increments the network frame number, which is used to synchronize actions and log files between the server and clients.
3. Loops through the packets received since the last frame, using a NetworkUtils function. The packets are handled according to type, which is detailed later in this section.
4. Compiles all world-state updates into an update packet, and sends it to all connected clients.
5. If the user has requested, send updated player configs to all connected clients.

The code that does this is outlined below:

```
/* FPSciServerApp.cpp */
void FPSciServerApp::onNetwork() {
    Framenumber ++
    while (there are packets to receive) {
        Receive a packet
        if (we received an unreliable packet) {
            handleUnreliablePacket(packet);
        } else {
```

```
            handleReliablePacket(packet);
        }
    }
    Create an update packet that has all player positions in it
    Broadcast the update packet to all connected clients

    if (the user has requested to propagate PlayerConfigs) {
        Send player config to the client(s) requested
    }
}
```

Listing 3.1: Basic structure of the onNetwork function. This function is called once per frame by the oneFrame function. It handles all updates from the network and sends updates the the clients.

There are three different packet types that the server expects to come on the unreliable channel:

- HANDSHAKE packets prompt the server to reply with a handshake reply packet to indicate to the client that the pure UDP socket is operational and connected.
- BATCH_ENTITY_UPDATE packets are processed by the server reading the updates in the packet and applying the update to the specified entity based on the update type. Currently, the client only sends an update for itself in the batch entity update but we designed the server to support clients sending updates for multiple entities in a single packet for future features. Similarly, we built-in support for different update types that can be specified on a per-update basis which is designed to provide support for easily switching between absolute updates, like we are using now, and relative updates such as deltas or any other representation of update data.
- PLAYER_INTERACT packets are sent to the server when a client does specific actions such as shooting (regardless of if they hit anything). At the moment all that the server does with this information is log it to the database so that researchers can review accuracy and fire-rate statistics for all players from a centralized file. Future versions of the game might broadcast these actions to other clients so that they can play a sound when another player shoots their gun and possibly render a bullet hole where the missed shot hit the scene.

```
/* FPSciServerApp.cpp */
void handleUnreliablePacket(packet){
    switch (packet type) {
    case HANDSHAKE:
        Send response to client
        break;
    case BATCH_ENTITY_UPDATE:
        for (each update) {
           switch (Update type) {
                case NOOP:
```

```
                    // Do nothing (No-Op)
                    break;
                case REPLACE_FRAME:
                    Update the entity's position from network
                    break;
            }
        }
        break;
    case PLAYER_INTERACT:
        Log remote action
        break;
}
```

Listing 3.2: This is the logic that is run when an unreliable packet is received by the server.

There are five packet types that we handle on the reliable channel, most of which are control packets that deal with connection and round management.

The first type of packet that we handle is a connection on the reliable channel, for these packets, we simply accept the connection and the ENetPeer structure will take care of sending a response back to the client indicating that we are connected.

If the packet is a disconnection on the reliable channel, we start by removing the client's model from our local copy of the world state and deleting their connection information. We then broadcast a message to all the rest of the clients that are connected to tell them to remove the disconnected client from their world as well.

If we receive a register client packet on the reliable channel, we store their connection information including their ID, unreliable port number, reliable ENetPeer object, and current frame number. We use this information in other parts of the game logic to determine the address or peer whom we should send a packet to based on their ID. After we have stored their information we add them to our local world state by creating a model and inserting it into our scene with their ID. Finally, we must synchronize this new world state with all of the clients, including the newly connected client. To do this, we first send a broadcast message to all the previously connected clients telling them to add a new entity that will represent the new client. Then, the server tells the new client to add each of the existing clients to their scene. Finally, the server tells the new client where their spawn position should be and tells them to respawn to that position now.

If the server receives a packet reporting a hit, it first logs the hit to the database, then it deals damage to the entity that was shot and checks if that shot destroyed the entity. If the shot did destroy the entity, the server commands all clients to respawn, resets the state of the round, and tells all the clients that a player has shot another player. If this shot does not destroy an entity, the server simply notifies all clients that a player has shot another player.

The final type of reliable packet that the server handles is a ready-

up packet. This packet is sent from the client when the user has indicated that they are ready for the round to start. When a server receives this type of packet, it simply increments a counter of how many clients are ready and if that counter reaches the threshold for the number of players needed to start a round it broadcasts a packet to all clients telling them to synchronize their frame numbers and start the round.

```
/* FPSciServerApp.cpp */
void handleReliablePacket(packet) {
    /* This is a reliable packet */
    switch (inPacket->type()) {
    case RELIABLE_CONNECT:
        Accept the incoming connection
        break;
    case RELIABLE_DISCONNECT:
        Delete the player from the scene
        Remove the their connection information
        Tell all the other clients to remove them from their scene
        break;
    case REGISTER_CLIENT:
        Record their connection information
        Acknowledge their connection
        Set the latency on their connection to the default value

        Create a model for the new client and add them to the scene
        Tell all currently connected clients to add the new entity
        to their scene
        for (each currently connected client) {
            Tell the new client to add existing client to their
            scene
        }
        Set the new clients spawn position
        Tell the new client to respawn
        break;
    case REPORT_HIT:
        Log the hit
        if (this hit kills a player) {
            Reset the game state
            Tell all clients to respawn
        }
        Tell all the clients that someone was shot
        break;
    case READY_UP_CLIENT:
        Players ready ++
        if (enough players are ready){
            Start the round and tell all clients to start
        }
        break;
}
```

Listing 3.3: This is the logic that is applied when the server receives a reliable packet.

Client-Side onNetwork

The client-side onNetwork implementation is largely the same as the server side, but it only deals with a single remote connection (to the server), and it has much less information to keep track of. The function does the following:

1. Calls the parent's onNetwork function. This does nothing as far as we can tell, but is good practice in case GApp default network functionality is added.
2. Increments the network frame number variable
3. If the unreliable socket is not connected, it crafts and sends an unreliable handshake packet to the server, hoping for a response.
4. If both channels to the server are open, it creates and configures a BatchEntityUpdate packet, fills in the location data of this client, and then sends it to the server.

```
/* FPSciApp.cpp */
if (reliable and unreliable channels are connected) {
    Create packet of updates
    Populate packet with data from player
    Send packet
}

```

Listing 3.4: This logic creates an update for the clients position and sends it to the server and is run once per frame.

5. Finally, it uses NetworkUtils to receive every packet that arrived since the last frame, responding to each in kind:
   If the packet was received on the ***un****reliable* channel:

```
/* FPSciApp.cpp */
switch (packet type) {
    case BATCH_ENTITY_UPDATE: {
        /* Take a set of entity updates from the server
        and apply them to local entities */
        for (Entity update in packet) {
            if (Update is not for this client) {
                Get entity object from scene
                if (entity is null) {
                    Log that the entity does not exist
                }
                else {
                    switch (update type) {
                    case NOOP:
                        // Do nothing (No-Op)
                    case REPLACE_FRAME:
                        Move the entity to new position
                    }
                }
            }
```

```
        }
    }
    case HANDSHAKE_REPLY: {
        Set unreliable channel connected to true
    }
    case PLAYER_INTERACT: {
        if (Player performing action is not this client) {
            Log action, state, time and player's position
        }
    }
    }
}
```

Listing 3.5: This is the logic that handles unrelaible packets on the client and is run from the onNetwork function.

The three types of packets received on the unreliable channel are processed as follows:

- BATCH_ENTITY_UPDATE: Loop through the contained updates, when an update matches a NetworkedEntity in the client's scene, update that entity's position
- HANDSHAKE_REPLY: Log the reply and store that the socket connected
- PLAYER_INTERACT: If the interacting player is not this client (i.e. is remote), log the player action

If the packet was received on the *reliable* channel:

```
/* FPSciApp.cpp */
    switch(packet type)){
    /* Handle Reliable packets here */
    case RELIABLE_CONNECT: {
        Get local unreliable socket port
        Print connection information
        Create a registration packet
        Populate the registration packet with this client's
            GUID and unreliable socket port
        Send the packet
    }
    case CREATE_ENTITY: {
        if (GUID in the packet != this client's GUID) {
            Load a player model
            Create an entity for the new player
            Insert new entity into the scene
            Add the new entity to hittable target list
        }
    }
    case CLIENT_REGISTRATION_REPLY: {
        if (response GUID == this client's GUID) {
            if (packet's status field is success) {
                Set reliable connection connected to true
                Set the latency of the reliable network
                    connection according to the session config
                Set the latency of the unreliable network
```

```
                    connection according to the session config
            }
            else {
                Log that the connection was refused
            }
        }
    }
    case MOVE_CLIENT: {
        Get this client's player entity
        Set its position according to packet
    }
    case DESTROY_ENTITY: {
        Get the specified entity from the scene
        Remove the entity from the scene
    }
    case SET_SPAWN_LOCATION: {
        Get this client's player entity
        Set its spawn position according to packet content
        Set its spawn heading according to packet content
    }
    case RESPAWN_CLIENT: {
        Get player entity from scene
        Call player's respawn method
        Reset the network session
    }
    case START_NETWORKED_SESSION: {
        Start the networked session
        Set the network framenumber to the value in the packet
    }
    case SEND_PLAYER_CONFIG: {
        Set session config values to those in the packet
    }
    default:
        Print a warning that an invalid type was received
    }
}
```

Listing 3.6: This is the logic that handles relaible packets on the client and is run from the onNetwork function.

3.4.3 Supporting Infrastructure

NetworkUtils

The NetworkUtils class provides utility functions for sending and receiving packets, controlling connection latency, and managing client connections. Because these functions do not have any game logic in them, they are shared between the server and client code. For sending packets, NetworkUtils provides four functions: broadcastReliable, broadcastUnreliable, send, and sendPacketDelayed. For receiving a packet, the class provides one function: receivePacket. For managing connection latency, the class provides 3 functions: setAddressLatency, removeAddressLatency, and setDefaultLatency. Finally, for managing client connections it provides one function: registerClient.

```
/* NetworkUtils.cpp */
void broadcastReliable(packet, enetHost) {
    for (each peer connected to the host) {
        if (the peer is connected) {
            Clone the packet to send
            Set the destination of the packet and mark it reliable
            send(packet);
        }
    }
}

void broadcastUnreliable(packet, srcSocket, dstAddresses) {
    for (each address in dstAddresses) {
        Clone the packet to send
        Set the destination of the packet and mark it reliable
       send(packet);
    }
}
void send(packet)
{
    Get the latency for the destination of this packet
    if (latency == 0) {
        Send the packet on the wire now
    }
    else {
       /* call the delayed send function */
        sendPacketDelayed(packet, latency);
    }
}
void sendPacketDelayed(packet, latency) {
    Get the current time
    Calculate the time to send with the current time + the latency
    Create a LatentPacket with the given packet and time to send
    Add the packet to the LatentNetwork queue
}
```

Listing 3.7: The NetworkUtils functions are called by the client and server app from the onNetwork function to handle sending packets.

Both of the broadcast functions provide very similar functionality but take slightly different inputs. Each duplicates the given packet, creating a copy for each destination. They configure the packets to send on either the reliable channel or unreliable channel, respectively. Both then pass all the packets to the NetworkUtils send function.

The NetworkUtils send function first checks the configured latency for transmitting to the packet's destination. If the latency is 0, it calls the packet's send function, sending the packet to ENet immediately. Otherwise, it calls sendPacketDelayed to send the packet after a delay. The sendPacketDelayed function finds the time the packet will be sent (now + the latency), creates a LatentPacket with that information, and passes it to the LatentNetwork thread.

```
/* NetworkUtils.cpp */
shared_ptr<GenericPacket> receivePacket(enetHost, socket) {
    if (there are packets to receive on the unreliable channel) {
```

```
4           Create a GenericPacket that will handle reading the packet
    type
5           Convert generic packet to a typed packet with the packet
    contents
6           Set the packet as unreliable
7           return packet;
8        }
9       if (there are packets to receive on the reliable channel) {
10           if (a client has connected on the reliable channel){
11               Create a relibaleConnect packet
12               Set the packet as reliable
13           }
14           else if (a client has disconnected on the reliable channel
    ) {
15               Create a reliable Disconnect packet
16               Set the packet as reliable
17               break;
18           }
19           else { /* we just received data on the reliable channel */
20              Create a generic packet to read the packet type
21               Convert generic packet to a typed packet with the
    packet contents
22              Set the packet as reliable
23           }
24           return packet;
25      }
26      /* No new packets to receive */
27      return nullptr;
28  }
```

Listing 3.8: The receivePacket function in network utils is called by the onNetwork function of the client or server in order check if there are any packets to receive.

The receive packet function takes both an ENetHost and a socket that should be checked for incoming packets. The first thing that this function does is check if there are any packets waiting to be received on the unreliable socket. If there is one, it receives the packet as a generic packet, checks the type, then, based on the packet type, it creates a typed packet that interprets the input correctly based on the packet type. It then returns the typed packet.

If there are no unreliable packets to be received when the receivePacket function is called, it services the reliable connection with an ENet function call. This ENet call serves to generate an event for any connection, disconnection, or packet delivery that has occurred on the reliable connection since the last call and also allows the host to send any messages that have been queued for sending. If there has been a connection on the reliable channel, this function generates a packet representing the new connection and returns that packet. Similarly, if there is a disconnection on the reliable channel, it generates and returns a packet. Finally, if there was a packet to receive on the reliable channel, the function does the same thing as it does with unreliable packets, creating a generic packet and then creating a typed packet containing the data from the packet. If there are no packets to receive the function indicates this to the caller

by returning a null pointer.

```
/* NetworkUtils.cpp */
void NetworkUtils::setAddressLatency(addr, latency)
{
    Remove any existing latency for the addr
    Add the address to the latency map for storage
}
void NetworkUtils::removeAddressLatency(addr)
{
    Remove any existing latency for addr
}
void NetworkUtils::setDefaultLatency(latency)
{
    Sets the default amount of latency
}
```

Listing 3.9: This is the logic to change the amount of latency that will be applied to a specifc address or the default amount.

The functions for managing latency are simple and straightforward. Latency is tracked in a C++ map object indexed by the destination address. This allows us to quickly look up the amount of latency to add to a packet based on the destination address. By specifying latency on a per-address basis we are able to specify different amounts of latency for 2 clients running on the same machine (same IP) as well as between the reliable and unreliable channels for the same client, because they operate on different ports. The setAddressLatency function works very simply by deleting any existing entry for the specified address and then adding the new latency to the map. This function is called on the server when a client connects, when a session is initialized on either the server or client, and when a player config is pushed down from the server. The removeAddressLatency function works by simply removing the address from the map if it is in the map. If the latency is not set after the entry is removed, the default latency will be used instead. Finally, the setDefaultLatency function simply changes the value of the defaultLatency member variable. This is used by the send function in the case that the destination address of the packet cannot be found in the latency map.

```
/* NetworkUtils.cpp */
NetworkUtils::ConnectedClient* NetworkUtils::registerClient(packet)
     {
    Create a new ConnectedClient
    Record the peer associated with the new client
    Record the ID associated with the new client
    Record the unreliableAddress associated with the new client
    return newClient;
}
```

Listing 3.10: This it the logic for creating a new ConnectedClient struct that contains all the information needed to identify the client. This is called by the Server's onNetwork function when a new client connects.

NetworkUtils provides a struct called ConnectedClient that contains

all the information to identify a client. This struct includes the ENetPeer that is used to communicate with them reliably, an ENetAddress that holds the destination for unreliable connections, the ID of the client, and the frame number for that client. The register client function creates and returns a new ConnectedClient that contains all their contact information, which is derived from the packet contents. This is then used by onNetwork to identify the ID of the client so that it can update the right entity and can translate messages from the reliable to the unreliable channel and vice versa.

Packets

We designed custom data structures to hold the information we send over the Internet. For convenience, every type of packet we use is a different class, but all are subclasses of GenericPacket. GenericPacket offers four constructors: one for broadcast packets, one for received packets (which takes as an argument and parses the received bitstring), one for packets on the reliable channel, and one for packets on the unreliable channel. These set the internal destination fields so that the created packet will be sent to the proper place over the proper channel.

GenericPacket also has functions for determining which channel it was sent on, whether it's inbound or outbound, its source and destination, and its type. There's also a clone function to copy the packet object.

Non-generic packets add data fields and override three methods: populate, serialize, and deserialize. Populate is used to place values into a packet once it is initialized; the arguments vary by what data that packet type carries. Serialize takes each of the packet's internal data fields and writes them into a binary object that can be sent over the network. Deserialize does the opposite, reading the data fields off of a binary object and into the object's fields. Deserialize is called automatically on the arguments for the createRecieve method, which creates an inbound packet object on a host when it receives a packet. The structure of the class harirachy is outlined in Figure 3.3.

```
/* Packet.cpp */
int GenericPacket::send() {
    Serialize this packet into a binary output
    if (this packet is reliable) {
        Create an ENetPacket that contains the binary to be sent
        Send the packet on the wire
       return the status code
    }
    else {
        Add the binary to a buffer
        Send the packet on the wire
       return the status code
    }
}

void GenericPacket::serialize(outBuffer) {
```

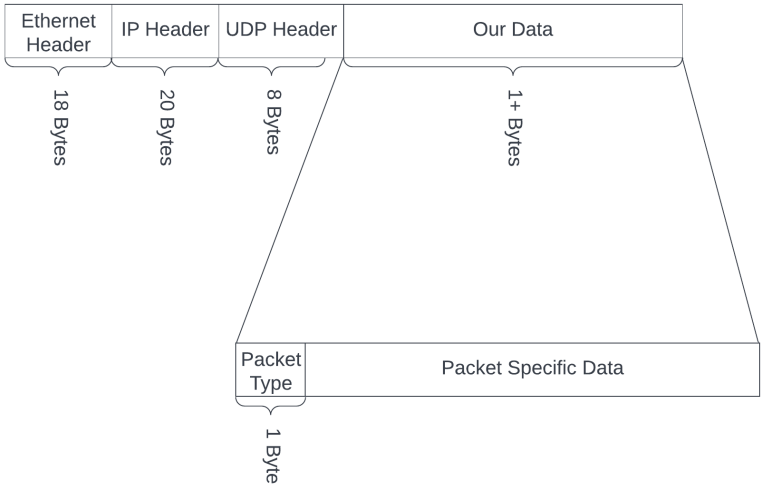


Figure 3.2: A GenericPacket starts with a packet type which then tells the receiving code in NetworkUtils how to interpret the rest of the data in the packet. This same structure is followed by all other packet types.

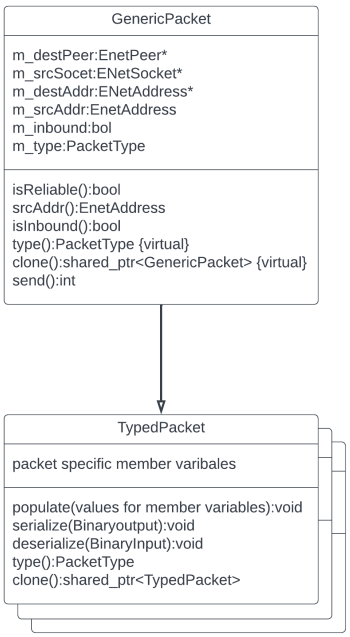
```
17      Write the packet type to the binary output
18  }
19
20  void GenericPacket::deserialize(inBuffer) {
21      Read the packet type from the binary input
22  }
```

Listing 3.11: these are some of the basic functions that are provided by the GenericPacket class. Derivied classes will override and implement their own verison of serialize and deserialize that will handle the specifc data in that packet.

We defined and used the following packet types:

- UNINTALIZED_TYPE: default value for uninitialized packets. Never sent over the network.
- BATCH_ENTITY_UPDATE: contains position and heading data for any number of players. Sent to the server from each client containing their own position, and from the server to every client containing every client's position.
- CREATE_ENTITY: instructs a client to create a new player entity, sent

Figure 3.3: A UML diagram of the packet class and derived packet types



by the server when a new player joins

- DESTROY_ENTITY: instructs a client to destroy a player entity, sent by the server when a player disconnects.
- MOVE_CLIENT: instructs a client to move its player to a new location. Only sent by the server.
- REGISTER_CLIENT: packet carrying information about a client. Sent to the server when a connection is first established.
- CLIENT_REGISTRATION_REPLY: sent to a client in response to a register request. It either can signal approval or denial of the registration request.
- HANDSHAKE: ask for a response.
- HANDSHAKE_REPLY: reply to a handshake request.
- REPORT_HIT: inform the receiver that a player has been hit and by whom. Sent from a client to the server, and then echoed to other clients.
- SET_SPAWN_LOCATION: instruct a client to set its player's spawn location to a specified value.

- RESPAWN_CLIENT: instruct a client to call its player's "respawn" method. Sent from the server.
- READY_UP_CLIENT: inform the server that this client is ready to start. Sent from the client.
- START_NETWORKED_SESSION: instruct a client to start it's networked session. Sent from the server.
- PLAYER_INTERACT: sent to log any form of player interaction, including moving, looking, and shooting. Sent from clients to the server, and echoed to other clients.
- SEND_PLAYER_CONFIG: carries most of the PlayerConfig data structure, sent from server to client as part of the configuration for a session.
- RELIABLE_CONNECT: created when a reliable connection successfully opens. Only used locally.
- RELIABLE_DISCONNECT: created when a reliable connection is closed. Only used locally.

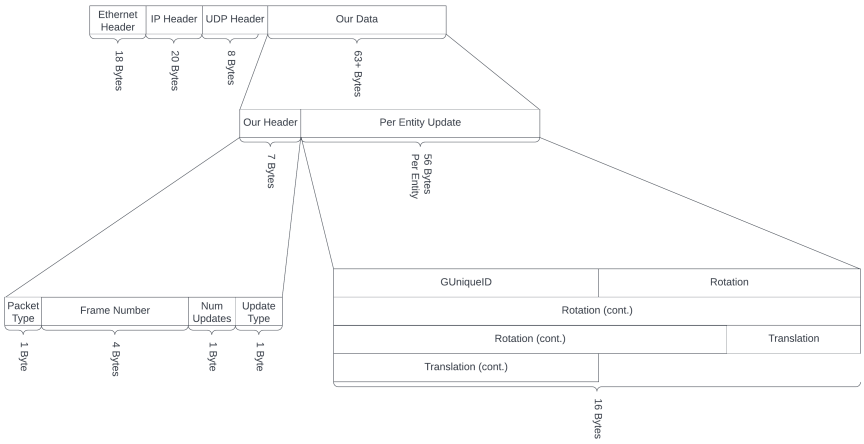


Figure 3.4: The structure of a BatchEntityUpdate packet (our most frequently sent packet) follows the structure of a GenericPacket but adds additional values to our header and then adds update information for one or more entities

LatenetNetwork

The LatentNetwork class provides functionality for simulating latency. This allows us to run experiments in a test lab with extremely low latency

and then create a specific, consistent amount of latency in the system. LatentNetwork runs an independent thread that polls a heap of LatentPackets every 0.5ms, sorted by their scheduled time to send. If the current time matches the time a LatentPacket is scheduled to be sent, it pops the packet from the heap and sends it. LatentNetwork also provides thread-safe access to add elements to the LatentPacket heap via a shared, mutex-protected queue. LatentNetwork offers one public function: enqueuePacket.

```
void enqueuePacket(shared_ptr<LatentPacket> packet) {
    {
        Lock the packet queue
        Push the packet onto the queue
        Unlock the packet queue
    }
};
```

Listing 3.12: This is the logic that adds a LatentPacket to the queue shared between the main thread and the neworking thread. This is called by the onNetwork code to hand off a packet.

The network thread runs this:

```
void LatentNetwork::networkThreadTick()
{
    while (thread is running) {
        Sleep until 0.5ms since the last tick

        Lock the packet queue
        Copy the queue contents into local heap
            Empty packet queue
        Unlock the packet queue

        Place new packets into the local heap

        Get the current time
        while (the next packet in the heap should be sent) {
            Pop the next packet
            Send the packet
        }
    }
}
```

Listing 3.13: This is the logic that the networking tread runs every time it wakes up. This handles taking packets form the main thread and sending them on the network after the right amount of latency.

Chapter 4

Evaluation

To evaluate our contributions, we designed and ran several experiments to determine the efficiency, scalability, and resiliency of our networked version of FPSci. Our testing was centered around the performance of the game, as FPSci must run at extremely high framerates to be able to do experiments with latency. We also tested the performance of the system in extreme conditions - high latency, packet loss, and large numbers of connected clients - to see how well it would perform. These tests gave us insight into the strengths and weaknesses of the code and what experiments it would be useful for.

4.1 Methodology

We identified several variables to examine:

- **The amount of time spent in the onNetwork function.** This lets us know how much CPU time our network code uses each frame. If we compare this to other parts of the game loop - physics, user interaction, rendering - we can find the relative impact of the networking code.
- **The network traffic our code generates.** Measuring the amount of traffic our code generates, and seeing how this changes when the framerate or the number of clients changes, allows us to see how efficiently we use the network, and how many clients can participate in a single match on a given network.
- **The impact of latency and packet loss on the clients.** Comparing the world states of the clients to each other and the server at any given time lets us see how much imperfections in the network connection affect their synchronicity. This gives us an idea of how resilient our code is to

network disruptions, and we can compare the behaviour of our system to existing games with latency compensation techniques.

- **The accuracy of our simulated latency.** If it is to be used in experiments, we want to confirm that our latency simulation is accurate and effective.

We implemented several changes to the system in order to run our evaluation. Most of these changes consisted of adding more logging to be able to record variables we measured, but overall they do have a significant impact on performance.

We added three tables to the logging database:

- **The amount of time spent in different parts of the code.** Each frame, we logged the amount of time the CPU spent executing the onNetwork function, the simulation code, and the graphics code. To collect this data, we hooked into the existing Profiler in G3D, which FPSci and G3D were already using to profile sections of the code, including the onSimulation function and the graphics functions. We added a profiler event for our onNetwork code, and added code to export the values from the profiler into the database each frame. Unfortunately, the running profiler is rather costly. When we ran matches before enabling it for the evaluation, we were able to run the game at up to 500Hz. With the profiler enabled for the evaluation, FPSci could not run faster than around 200Hz.
- **The network traffic sent and recieved.** In order to measure the amount of network traffic our networking code generated, we hooked the logger into the NetworkUtils send function to count the number of bytes sent each frame. This byte count, along with the number of packets sent is logged each frame. The receive function in NetworkUtils also records the number of packets received and their total size in bytes. This provides a measure of the network traffic our code is generating, but unfortunately it does not account for all of the data. ENet's reliable channel periodically sends small heartbeat packets and adds header data and acknowledgement packets to the information we send through it. While this probably is not a significant amount of data, we can not measure it with our FPSci-based logging because it is at a lower layer than we have access to.
- **The timestamps ping packets are sent and received.** Periodically, we send ping packets (which we dubbed "serial number" or SN packets, because they contain only a serial number) from the clients to the server. By recording the exact frame and time the each packet is sent, and then the time it is echoed back, we can calculate the two-way latency in the network connection. We also record the time that these packets are processed by the server, but we were not able to do any analysis on one-way latency because we were not able to synchronize our test computers' clocks to a satisfactory precision.

To stress-test the server, we also created a "headless" client, which does no simulation and has no user interface. This client waits for the match to start then simply reads a recording of a previous match, and each frame sends the correct packet to simulate having moved. To the server, this is almost indistinguishable from a real client, although it does not respond to all of the control packets the server can send. Because this client is so lightweight, we can run many on a single machine and see how the server scales with many clients.

We setup three machines with identical specs to run our experiments. These machines each had an Intel i7-8700K at 3.7GHz with 64GB of ram, a GTX 1080 and an Intel I219-LM 1Gb/s NIC. These were connected together with a 1Gb/s unmanaged network switch which had a 1Gb/s connection to the local network. Two of these machines were used as clients and the third was used to run the server. Additionally, in order to run the headless client we used a machine with an AMD Ryzen 2600 at 4.1GHz with 32GB of ram, an RX 580 and an HP 560SFP+ 10Gb/s NIC connected to the local network at 10Gb/s.

Finally, to simulate adverse network connections, we used a Raspberry Pi as a router between the server and the local network. With the Pi, we were able to add packet loss on the connections between the clients and the server.

We devised four different sets of experiments to examine performace of our code:

- **Test 1: Traditional Matches with Variable Framerate**:

  The first set of trials we designed were aimed at getting a baseline of performance for our modified version of FPSci. We ran several trials with 2 clients playing against each other. Each trial lasted approximately 60 seconds. We ran each trial at a different framerate, although in each trial the framerate of both clients and the server matched. We used framerates at 30Hz intervals between 30Hz and 210Hz. Above 180Hz, FPSci clients were unable to maintain the requested framerate, so we did not continue testing higher framerates. When the profiler is not enabled, the game can reach over 400Hz consistently.

- **Test 2: Stress-Testing the Server with many Clients**

  The second test we performed was aimed at determining how many clients the server would be able to handle at any one time. We recorded a one-minute session of a client moving around the map, and then replayed that through increasing numbers of headless clients. We ran trials with 2, 4, 8, 16, 32, 64, 128, and 255 clients.

- **Test 3: Varying Latency and Examining its Effects**

  The third test we performed was aimed at determining how latency affects the synchronization of the clients. We ran several one-minute 60Hz trials with two clients, with the same, symmetric latency on each link. We increased the amount of latency by 20ms each trial. Inspecting the logs

allowed us to see how the measured latency compared to the specified latency, and how memory and CPU usage were affected by the increased latency. We were also able to compare at each point in time the position of each client's player on itself, on the server, and on the opposing client.

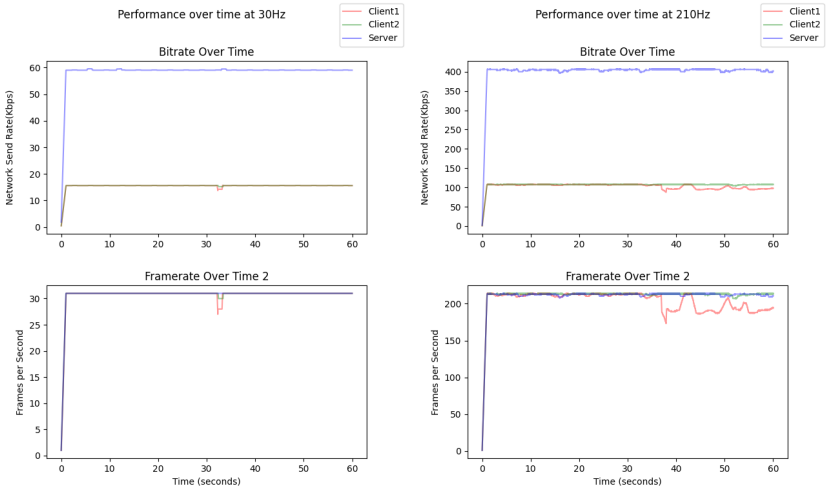
- **Test 4: Examining the Effects of Packet Loss**

  Finally, we used the Pi router to introduce packet loss into the system. We ran 2 player, 60Hz trials with no added latency, and with 0.1%, 0.25%, 0.5%, 1%, 2%, 5%, 10%, and 20% loss. We were watching for errors in the communication and how the discrepancy between the clients changed - how large it was and how consistent it was, and whether noticeable jitter appeared to the user.

4.2 Results and Analysis

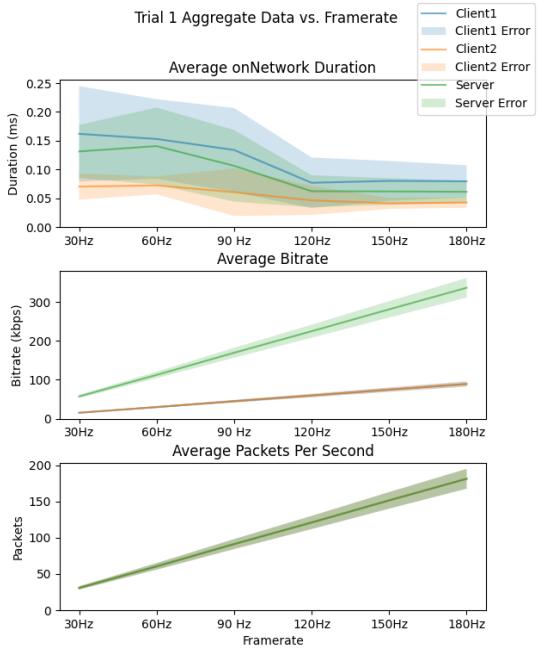
4.2.1 Test 1: Traditional Matches with Variable Framerate

Figure 4.1: Overview graphs of a trial run at 30Hz (left) and 210Hz (right)



As can be seen in some of our initial graphs from our first experiment in Figure 4.1, the bitrate and framerate are nearly constant. The small variations are likely from random changes in graphics time and are not significant compared to the average value. We found the bitrate, framerate, CPU time for onNetwork, and number of packets per second to be constant within a trial, allowing us to compare across trials using the average values from each trial. The graphs from the trial at 210Hz, where FPSci could not hold the requested framerate, show

Figure 4.2: Average bitrate, packets per second, and CPU time vs. framerate



significant variation in the client 1 framerate. We did not continue testing at higher framerates, and do not include the 210Hz data in our analysis, because at that point FPSci is operating outside of normal conditions.

One feature to note about the 210Hz graph is that as the framerate of the client decreases, so too does the network bitrate; the two are directly related. This relationship is confirmed when we look at Figure 4.2, the bitrate of the server and clients' network communication increases linearly with framerate. This is because FPSci sends the same amount of data each frame regardless of the framerate (about 63 bytes per frame from client to server), so the *per second* network throughput is higher when there are more frames each second

The server sends more data than either client because it is sending the state of *all* networked entities to *all* clients, whereas each client only sends its own position data to the server.

We cannot explain why the average time spent in onNetwork is greater at lower framerates for the server and client 1 during our experiments. It is possible that this is a result of random variation or a single very slow frame dragging the average, but the fact that the same curve is present on both a client and the server suggests this is not the case.

4.2.2 Test 2: Stress-Testing the Server with many Clients

We can predict the amount of data sent over the network based on the number of clients connected. Of all the types of packets we defined, only the BatchEntityUpdate packets have much affect on the network utilization. All other packet types are either extremely small or sent very rarely, or both. A BatchEntityUpdate consists of two sections: a header and a list of entities to be updated (refer to Figure 3.4 for update packet structure). Knowing that the header consists of 7 bytes and that one entity requires 56 bytes, we can find an equation for the bitrate of data sent by the server and clients. Given that F is the framerate in frames per second and n is the number of clients on the server,

$$bitrate_{client} = (7 + 56) * (F) * 8$$

$$bitrate_{server} = (n)(7 + (56)(n))(F) * 8 = 7(n)(F) + 56(n^2)(F) * 8$$

Using these equations, we can predict the network usage for trials based on framerate and number of clients connected:

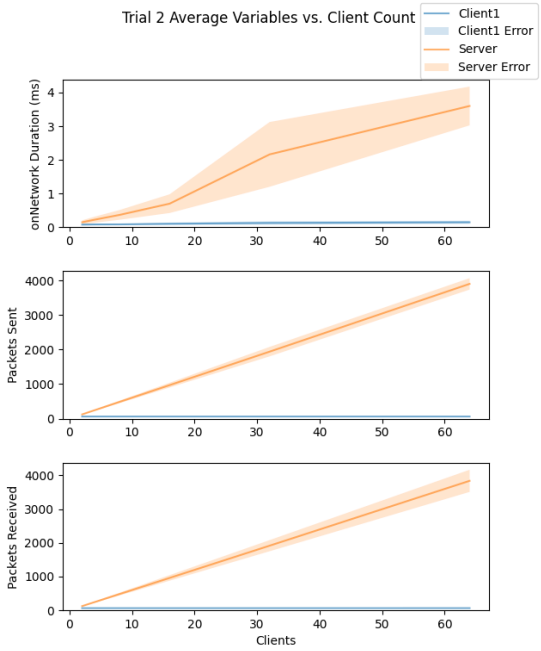
Expected Network Throughput at Different Client Counts (Mbps)

Hz	2	4	6	8	16	32	64	128	256
30	0.057	0.222	0.494	0.874	3.47	13.8	55.2	220	881
40	0.076	0.296	0.659	1.16	4.62	18.4	73.5	294	1175
50	0.095	0.370	0.823	1.46	5.78	23.0	91.9	367	1469
60	0.114	0.444	0.988	1.75	6.94	27.6	110	441	1762
70	0.133	0.517	1.15	2.04	8.09	32.2	129	514	2056
80	0.152	0.591	1.32	2.33	9.25	36.8	147	588	2350
90	0.171	0.665	1.48	2.62	10.4	41.4	165	661	2644
100	0.190	0.739	1.65	2.91	11.6	46.1	184	735	2937
110	0.209	0.813	1.81	3.20	12.7	50.7	202	808	3231
120	0.228	0.887	1.98	3.49	13.9	55.3	221	882	3525

Table 4.1: Predicted server network utilization (Mbps) at given client counts (x-axis) and framerates (y-axis)

Our measurements, as seen in Figure 4.4, almost exactly match the predictions in the Table 4.1: the server's network utilization increases exponentially with the number of clients. This is because adding more clients increases both the number of packets to send *and* the amount of data sent in each packet. Since clients only communicate with one other host, the server, their network utilization increases linearly with both framerate and client count. The data received by the server increases linearly as well, because each new client only sends a fixed amount of data. The most intensive experiments that can be accommodated by a given *network* is easily calculable from Table 4.1. On a 1-Gbps LAN, the most clients that can be serviced on a 60Hz server is around 128.

Figure 4.3: Graphs of average parameter per trial vs. number of clients

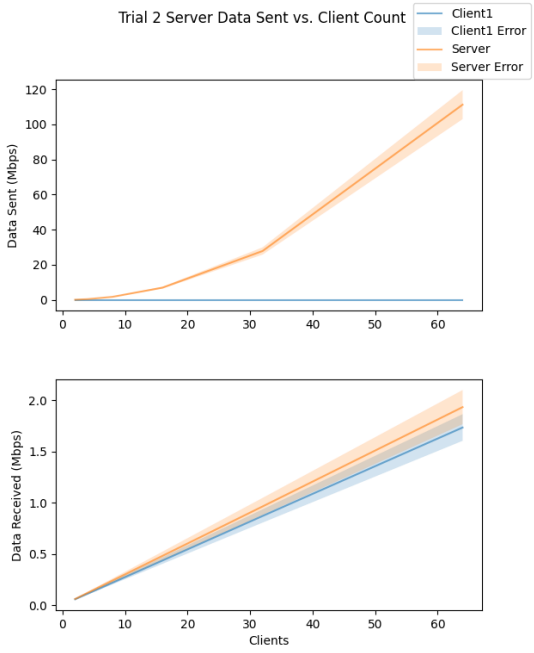


The server running on our test hardware could not handle 128 clients connected simultaneously, but did run smoothly with 64 clients. As can be seen in figure Figure 4.5, the total amount of server CPU time spent each frame exceeds the duration of one frame (about 16.6ms at 60Hz) somewhere between 64 and 128 clients. With more clients, server CPU time increases exponentially. Somewhere between 128 and 255 clients, the amount of time spent each frame on the network alone exceeds the duration of one tick.

On the client, as seen in Figure 4.6, the amount of CPU time used in the network code remains relatively constant. However, the amount of CPU time spent on graphics increases with the number of clients as the game is forced to render more player models. In terms of pure computation, a client can easily handle more than 254 other clients in the same match.

However, our software can currently only support 255 clients connected simultaneously because a batch entity update packet stores the number of entities it contains in a single byte. This could be fixed by either increasing the size of the entity count field, or by fragmenting updates across multiple packets when there are too many to fit in a single one. However, at higher client counts the processing cost and network cost of new clients is very large; 256 clients at 70FPS would easily saturate a 2Gbps link. Most experiments we foresee will need no more than a dozen clients at most, so our implementation is robust

Figure 4.4: Graphs of average network utilization (send and receive) vs. number of clients



enough for our use case.

4.2.3 Test 3: Varying Latency and Examining its Effects

We found that even when we requested no latency be added to the networking code we still observed around 16.6 ms of latency round trip from client to server. This is what we would expect when running the game at 60Hz: the client sends the packet to the server, the server then has to wake up from it's game loop, reply to the packet, and then the client has to wake up from its game loop and receive the packet. Waiting for each host to wake and service the network should add a delay of a considerable fraction of the frame time. We found that this amount of additional latency seems to be fairly constant regardless of the amount of latency that we configured the networking thread to add. As can be see in Figure 4.7, the amount of additional latency above the requested latency is fairly consistently between 16 and 20 ms which is just over one frame time's worth of latency. We then re-ran our analysis code on the data gathered as part of test 1 to see if at higher framerates there was lower additional latency, supporting the hypothesis that we should expect one frame of additional latency. Looking at Figure 4.8 we can see that this trend continues

Figure 4.5: Graph of server CPU time spent in different functions vs. number of clients

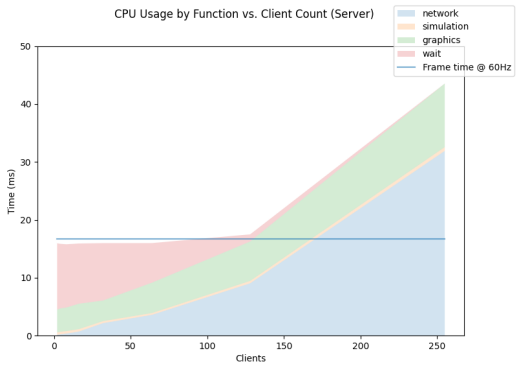
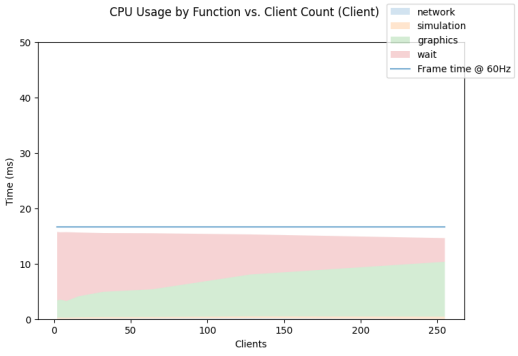


Figure 4.6: Graph of client CPU time spent in different functions vs. number of clients



as we would expect and the amount of additional latency is approximately equal to the frame time.

Additionally, we found that there appears to be a linear relationship between the deviation of position between local and remote clients and the amount of latency we added, as can be seen in Figure 4.9. This is what we expected because the trial that the headless client was replaying for this test had the client moving straight ahead almost constantly. This meant that as the local client lags further and further behind, the remote client would get further and further from where they were seen locally (since the distance the client moves in the time between its position is sent and received is directly proportional to the time between sending and receiving). We would expect that this is the upper bound of deviation because in a real game the player would not be moving consistently forward. Any time they are not moving, the lagging client's copy of their player would "catch up" to the unmoving player's actual

Figure 4.7: Graph of requested latency and the amount of additional latency measured.

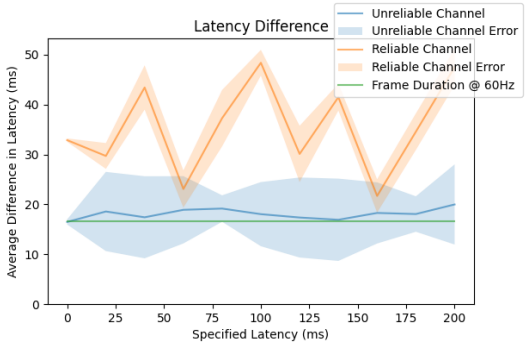


Figure 4.8: Amount of round trip latency observed at different framerates

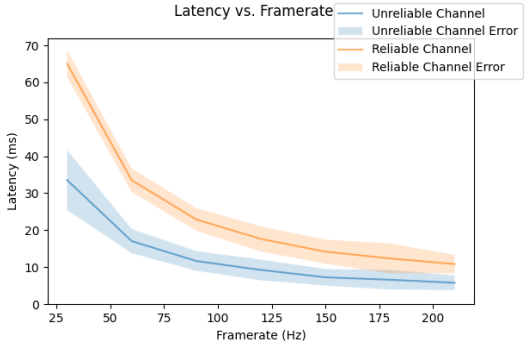


Figure 4.9: Deviation in position between clients at differing added amounts of latency

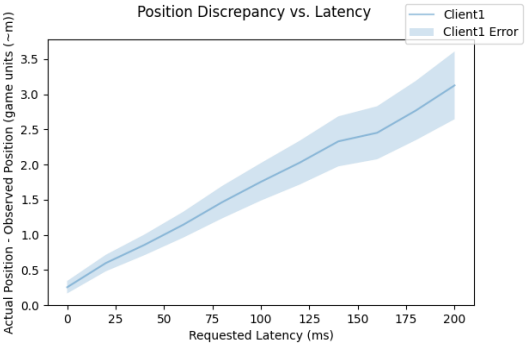


Figure 4.10: Deviation in position between clients during 2% packet loss

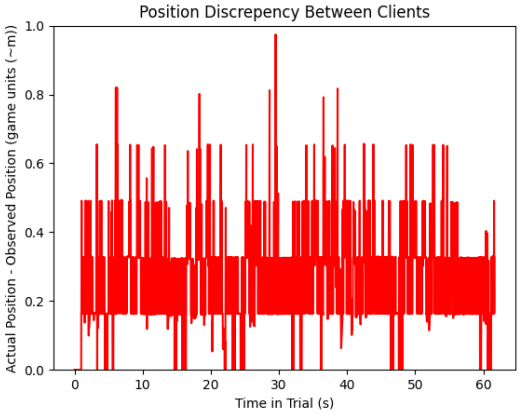
position, decreasing the deviation during certain parts of the match.

4.2.4 Test 4: Examining the Effects of Packet Loss

We found that packet loss has surprisingly little impact on our system. Our perception while playing was that packet loss was not noticeable until it was significant (over 2%), and then only manifested as slight jitter in other clients' movement. These jitters did not have enough magnitude to make aiming more difficult.

The reason packet loss has so little effect is that we are constantly sending position updates. That means that even if a packet is lost one frame, the next frame will most likely yield a packet with the correct location. Because we deal in absolute locations, there is no loss of location precision if this happens. Sending data at 60Hz means that the discrepancy is only present for about 16ms. In games with slower framerates (or slower network update rates), it would become a problem more quickly. We found that the discrepancy stays relatively low, as seen in Figure 4.10, but fluctuates between multiples of the the movement speed and frametime as packets are lost. Some of these fluctuations are from random variations in latency, but packet loss also contributes, as can be seen at higher loss rates in Figure 4.11. Most discrepancies are equivalent to only one lost frame, which makes sense in our experiment - the chance that two packets will be lost in a row is the loss rate squared, which was only 4% even in our 20% packet loss test. In real traffic, packet loss tends to occur in bursts, which would have a more noticeable effect. However, the game would recover as soon as communication resumes. It is worth noting that because networked FPSci is client-authoritative, any shots that hit the opponent while it appears stationary would still be registered as hits.

Figure 4.11: Deviation in position between clients during 20% packet loss



Chapter 5

Conclusion

Gaming, and especially first person shooter games, are a popular form of entertainment that comprise a sizeable portion of the entertainment industry [8]. The size of the market encourages the development of high-quality games to draw players, and to do this requires understanding how players and games interact. Additionally, the understanding gained by studying games can be applied to other areas of human computer interaction, improving the user experience in unrelated areas of computing. The impacts of latency and input delay are an important area of study in games, especially in first person shooters, where fast-paced play requires quick thinking and reaction times.

Our goal was to expand FPSci, a research first person shooter, by adding the ability to run networked multiplayer matches. On top of FPSci's existing ability to run experiments with local latency, this would facilitate research into the impact of different parameters on players' performance playing multiplayer games. As we were expanding upon an existing project, was important that we change as few things as possible so that FPSci could still be used for single-player experiments.We also kept the same system of using the Any files to specify parameters for experiments to align with the FPSci design of allowing easy, extensive configuration for experiments.

As well as performing all of the functions that singleplayer FPSci provides, our version allows multiple clients to connect to a server and play together. Each client is able to see all of the others in real time and constantly shares its own location data. All of the clients are also capable of shooting other players. FPSci detects successful hits and shares them with the server, which responds in kind. Currently, it restarts the match by respawning all of the players, although this could be modified easily to create other varieties of gameplay.

To facilitate experiments, the number of clients a server supports, the latency of connections, and server address and port are configurable through files. As of this writing, our code is being used in a WPI study measuring how

latency affects the peaker's advantage in multiplayer FPS games.

As demonstrated in our evaluation, the code we wrote can run successfully in adverse conditions, and, more importantly, can log characteristics of these conditions and its own performance for evaluation. We also left the code in a relatively readable and extensible state: the switch statements in onNetwork can easily be modified or extended to incorporate new packets. The generic packet infrastructure also means that adding new types of packets is simple and easily understandable, allowing new packets to be added quickly by handling boilerplate code. Overall, our code is organized and labeled (as we did several refactoring operations throughout implementation), making extending it easier.

Future Work

There are certainly more features that would benefit multiplayer FPSci. To make the experience more closely match FPS games, future developers can add game features, optimize code, or change the multiplayer architecture.

Possible new game features include different weapons, such as projectile launchers, items that players can collect, or other game modes. These could prove useful in some experiments and bring the game experience match more closely what users would encounter in the real world. Of course, each of these would need to be handled over the network to ensure projectiles follow the same path for all clients, only one client can pick up any single item, and managing different respawn behaviour or player organizations.

There are also ways to improve the performance of our code. It is possible to decrease the network utilization by compressing the data we send, filtering it to avoid sending repeat updates, or scheduling updates more intelligently. We currently send all of the available data every frame, without compression, regardless of whether any other instance needs it, which could be fixed with a modest amount of tweaking.

It should also be possible to increase server performance with a large number of clients by improving how references to the player objects. Our code relies on linear searches to find particular clients, storing references would improve lookup time significantly without having much affect on memory usage. It would also be useful to researchers to have an authoritative server and to have different styles of sending data (i.e. via deltas instead of absolute values), because these systems are commonly used in production games.

Bibliography

[1] B. Boudaoud, J. Spjut, and J. Kim, "Firstpersonscience: An open source tool for studying fps esports aiming," in *ACM SIGGRAPH 2022 Talks*, SIGGRAPH '22, (New York, NY, USA), Association for Computing Machinery, 2022.

[2] "First Person Science," May 2022. `https://github.com/NVlabs/FPSci`.

[3] M. McGuire, M. Mara, and Z. Majercik, "The G3D innovation engine," 01 2017. `https://casual-effects.com/g3d`.

[4] Y. Pisan, "Challenges for network computer games.," in *Proceedings of the IADIS International Conference WWW/Internet*, pp. 589–595, 01 2004.

[5] D. Bethea, R. A. Cochran, and M. K. Reiter, "Server-side verification of client behavior in online games," *ACM Trans. Inf. Syst. Secur.*, vol. 14, 12 2008.

[6] J. Spjut, B. Boudaoud, K. Binaee, J. Kim, A. Majercik, M. McGuire, D. Luebke, and J. Kim, "Latency of 30 ms benefits first person targeting tasks more than refresh rate above 60 hz," in *SIGGRAPH Asia 2019 Technical Briefs*, SA '19, (New York, NY, USA), p. 110–113, Association for Computing Machinery, 2019.

[7] W.-K. Lee and R. K. C. Chang, "Evaluation of Lag-related Configurations in First-person Shooter Games," in *International Workshop on Network and Systems Support for Games (NetGames)*, pp. 1–3, 2015.

[8] S. Liu, M. Claypool, A. Kuwahara, J. Sherman, and J. J. Scovell, "Lower is better? the effects of local latencies on competitive first-person shooter game players," in *Proceedings of the CHI Conference on Human Factors in Computing Systems*, CHI, (New York, NY, USA), Association for Computing Machinery, 2021.

[9] T. Henderson, "Latency and user behaviour on a multiplayer game server," in *Networked Group Communication* (J. Crowcroft and M. Hofmann, eds.), (Berlin, Heidelberg), pp. 1–13, Springer Berlin Heidelberg, 2001.

[10] J. B. Spjut, B. Boudaoud, K. Binaee, Z. Majercik, M. McGuire, and J. Kim, "FirstPersonScience: Quantifying psychophysics for first person shooter tasks," *ArXiv*, vol. abs/2202.06429, 2022.

[11] Z. Li, H. Melvin, R. Bruzgiene, P. Pocta, L. Skorin-Kapov, and A. Zgank, *Lag Compensation for First-Person Shooter Games in Cloud Gaming*, pp. 104–127. Cham: Springer International Publishing, 2018.

[12] S. Vlahovic, M. Suznjevic, and L. Skorin-Kapov, "Challenges in assessing network latency impact on qoe and in-game performance in vr first person shooter games," in *15th International Conference on Telecommunications (ConTEL)*, pp. 1–8, 2019.

[13] L. Salzman, "ENet," 11 2020. `http://enet.bespin.org/`.