Expertise
and insight
for the future

Olli Vilmi

Real-time Multiplayer Software Architecture

Metropolia University of Applied Sciences

Bachelor of Engineering

Information and Communications Technology

Bachelor's Thesis

13 March 2020

Author Title Number of Pages	Olli Vilmi Real-time Multiplayer Software Architecture 49 pages + 2 appendices
Degree	Bachelor of Engineering
Degree Programme	Information and Communications Technology
Professional Major	Software Engineering
Instructors	Janne Salonen, Head of Department

The goal of the project was to study real-time multiplayer programming to identify requirements, problems and common solutions. Using this information, software architecture was designed and implemented for a 2D platformer game.

The programming language Lua was the programming language used for the game client, which runs on the LÖVE2D game framework. Consequently, the server was also written in Lua, but any applicable language could have been chosen. The freedom to be able to use any tools was an important objective of the project.

The server model, updating the game state, bandwidth and latency are critical topics which were addressed during the design process. The desired features for networking were implemented with simple solutions based on different requirements.

Experiences and problems faced during the development are reviewed to show examples of problems and to draw conclusions. In order to prevent packet loss without inducing more latency, features from TCP were implemented on top of a UDP based messaging protocol. Bandwidth usage was reduced by limiting world updates to the visible area for each player.

Keywords	multiplayer game, architecture, programming patterns, latency, networking

Tekijä Otsikko Sivumäärä Aika	Olli Vilmi Real-time Multiplayer Software Architecture 49 sivua + 2 liitettä 13.3.2020
Tutkinto	insinööri (AMK)
Tutkinto-ohjelma	Bachelor of Engineering
Ammatillinen pääaine	Software Engineering
Ohjaajat	Osaamisaluepäällikkö Janne Salonen

Projektin tavoitteena oli tutkia reaaliaikaisen moninpelin ohjelmoimista, jotta olisi mahdollista ymmärtää yleiset vaatimukset, ongelmat sekä ratkaisut. Lopputuloksena ohjelmisto suunniteltiin sekä toteutettiin kaksiulotteista tasohyppelypeliä varten.

Pelimoottoriksi valittiin LÖVE2D ja sen ohessa ohjelmointikielenä toimi Lua, mutta mikä vaan sopiva ohjelmointikieli tai pelimoottori olisi voitu valita. Suunnitteluvaiheessa tärkeä tavoite oli riippumattomuus työvälineistä.

Serverimalli, pelitilan päivittäminen, kaistanleveys sekä viive olivat kriittisiä aiheita, joita käsiteltiin suunnitteluprosessin aikana. Viestintäprotokollaa muokattiin tarpeiden mukaan toteuttamaan vaatimukset mahdollisimman yksinkertaisilla ratkaisuilla.

Kokemuksia ja ongelmia kehityksen aikana näytetään esimerkkeinä. Jotta olisi mahdollista estää tietopakettien katoamista aiheuttamatta lisää viivettä, ominaisuuksia TCP:stä toteutettiin UDP-pohjaiseen viestintäprotokollaan. Kaistanleveyden käyttöä vähennettiin rajaamalla maailman päivitykset pelkästään pelaajalle näkyvään alueeseen.

Avainsanat	moninpeli, arkkitehtuuri, ohjelmointisuunnittelumalli, viive, tietoverkko

Contents

List of Abbreviations

DNS	Domain Name Server
I/O	Input and output
ID	Identifier
IP	Internet Protocol
JSON	JavaScript Object Notation
LAN	Local Area Network
LZW	Lempel-Ziv-Welch compression algorithm
MD5	Message-digest algorithm
MMO	Massively Multiplayer Online
MTU	Maximum Transmission Unit
NAT	Network Address Translation
P2P	Peer-to-peer
RPC	Remote Procedure Call
RTT	Round trip time
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
WAN	Wide Area Network
XML	Extensible Markup Language

1 Introduction

Real-time online multiplayer games are becoming the norm, to the extent that single player games are can only be played online. Multiplayer games are rarely developed by indie developers due to the added complexity, work and problems networking causes. It is a great risk because players may become easily frustrated if the game does not feel responsive.

Developers have come up with different techniques over the years that improve responsiveness and handle unstable connections. Yet, even the largest companies make mistakes in the pitfalls that it may feel as if the expectations are impossible to meet for an indie developer.

Due to limitations in hardware and network connections, different games require different solutions to optimize the performance of multiplayer games. This study investigates the theory behind reliable real-time multiplayer games, describing how a functional multiplayer architecture can be designed and developed.

The topic was chosen because credible sources are sparse. As a result, most information on the internet is scattered among blogs and message boards. In order to piece the information together, prior knowledge of networking and game development is required. The goal was to research the most essential steps of building online games, by building the networking solution from the ground up by only using basic network libraries that are available on any platform. Simple solutions are analyzed to find possibilities to accelerate the speed of development, which is especially important for an indie developer.

2 Theory

2.1 Context

The goal of the thesis was to design and implement a network architecture which allows the player to connect, move around, and affect the game world in real time. An arena shooter game is developed, where the player can build and destroy tiles to update the world and shoot bullets at other players. The game was inspired by the combination of an old game called Liero Extreme and a popular indie game called Terraria.

As a starting point, it was necessary to research game programming patterns and networking models in order to understand the requirements. The game's networking implementation was to be based on the solutions that were used in previous successful video game titles and books related to the subject.

An additional challenge for the project was updating the game world, which may contain more than 100,000 tiles without suffering from severe performance issues. Therefore, methods to improve performance were to be researched.

2.2 Game loop

A game loop is a program loop that updates a game as often as the *frame rate* allows it. Cycles of the loop are called *frames* (a single execution of the loop is one frame). The framerate is typically restricted to the refresh rate of the monitor or uncapped so that the update speed is only limited by the hardware. [7, ch. 3.9.]

The game loop typically consists of game state, which is the simulation of entities moving in the world, collisions, events and so forth. In a multiplayer game, the loop is typically run on a *headless server* (no graphic display), with clients sending input and requesting updates. The game loop will continue even when all clients disconnect. [9, ch. 6.]

In the 1990s, programmers were able to develop games with the expectation that a game loop would execute 30 times per second. This means that the game would be dependent

on the hardware being fast enough to process each frame. This was somewhat plausible because consoles shared the same hardware, but in reality some frames take more time to process.

In more complicated games it is much more difficult to ensure a stable framerate, which results in unpredictable frame rate drops**,** which would also cause the game execution speed to slow down**.** Gaming consoles were able to solve the issue by limiting the frame rate at a lower value than the hardware is capable of to ensure more stability. [7, ch. 3.9.]

When it comes to personal computers, this matter is even worse because the hardware is not the same, causing massive discrepancies in framerates across different devices. A device with strong hardware and uncapped frame rate effectively makes the game execute faster than what was intended, which affects the simulation of physics such as gravity. [7, ch. 3.9.]

A better solution is updating the game by *delta time*, the amount of time passed since the previous frame. This allows batching the updates, and a server which is updating the game loop 20 times per second ends up in the same state as the client updating the loop 60 times per second. The game loop is illustrated in Figure 1.

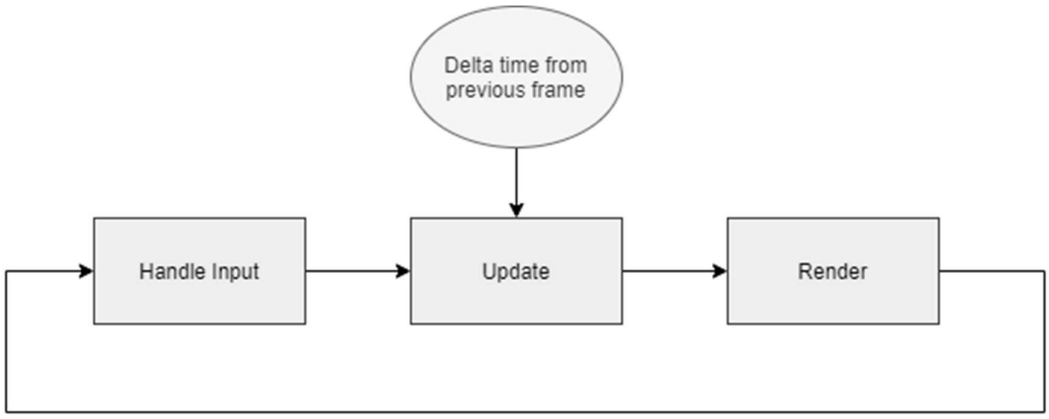

Figure 1. Game loop [7, ch. 3.9]

To provide an example for Figure 1, the client receives input to move left, the player is moved based on the amount of time passed, and the updated state is drawn to the display. [7, ch. 3.9.]

2.3 Multiplayer server models

In the Client-Server model, there are two ways of handling state changes on the server. The first one is the *server-authoritative model***,** which means that the clients are trying their best to match the server's game state, which is the single point of authority on game state. The advantage this model has is cheat detection, as the server only accepts user inputs and returns *snapshots* of the game state. [14.]

However, the disadvantage comes in the form of latency. An authoritative server is always ahead of the clients, which causes synchronization issues that are further discussed in chapter 2.6. In the 1990s, game clients for fast paced shooters such as Quake (1996) were designed as *dumb terminals*, which were not designed to work over the internet. [9, ch. 1.]

The server follows a *non-authoritative model* if the server is not the single point of authority**.** In this case, the client also simulates the game state. The client may move on their own and send updates to the server about the clients' events, such as colliding with a bullet for the server to handle. The benefit is that it is simple to implement, but makes cheating easier as the client has full control over all player movement. [17.]

It is plausible to have both server-authoritative and client-authoritative events. For example, player movement can be client-authoritative and bullet collisions server-authoritative.

2.4 Network topologies

Some game studios maintain their own servers, depending on the business case. Maintaining servers is expensive, which is the main consideration when it comes to choosing

which topology to use. Hosting a game server requires considerably more upload speed which is often limited in cheap broadband connections.

This is usually not a problem for *dedicated servers*, because the minimum internet and hardware requirements can be estimated so it does not come as a surprise that the server is unable to handle the given load. Nonetheless, the server load increases linearly with the number of players so a server will not be able handle amounts that exceed the estimations [16].

Some genres, such as Massively Online Multiplayers (MMOs) must be run on dedicated servers to be able to support thousands of players. Furthermore, single player games are sometimes published online only to prevent pirating. Most real time games have a hard a limit on how many players may be connected at a given time. Thus, scaling the server for hundreds of players is beyond the scope of this thesis. [9, ch. 1.]

The *Client-Server model* consists of two major components, namely the client and the server**.** The server is a computer running a centralized service, and clients are computers which connect to the service via the internet. Generally, the client *requests* specific services from the server and the server provides a *response* for the requested service. The Client-Server model is depicted in Figure 2. [6.]

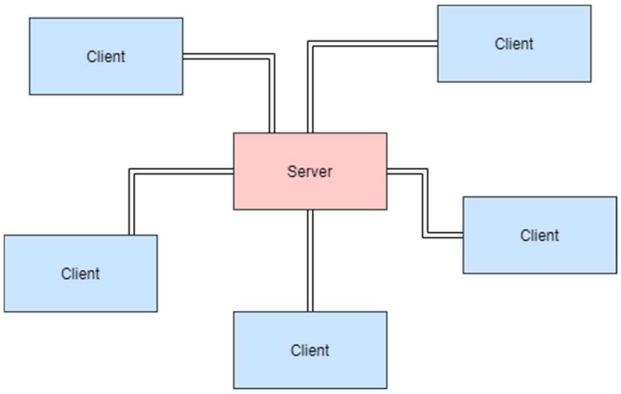

Figure 2. Client-server model [9, ch. 6]

As shown in Figure 2, the server scales linearly *O(2n)* for each client connection, sending and receiving data for each new client. However, the connection is *asymmetric*, because

each client only communicates with the server. This allows better performance for clients, which are often more limited in resources. [9, ch. 6.]

The alternative to a dedicated server is a *listen server*, which means that the *host* is also a player in the game. As a result, the disadvantage is that the machine must be powerful enough to host and play at the same time. Some games automatically set up a player as the host, which is not to be confused with the *Peer-to-Peer model* (P2P) which is shown in Figure 3. [9, ch. 6]

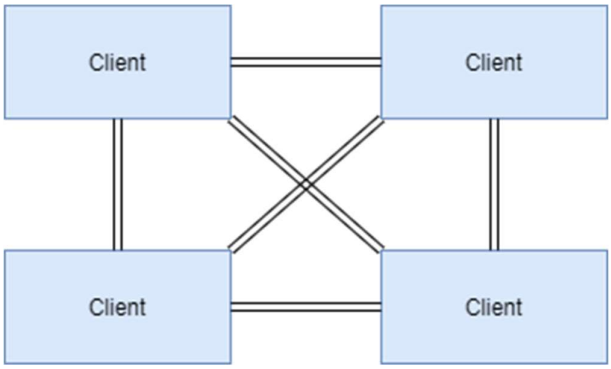

Figure 3. P2P model [9, ch. 6]

As shown in Figure 3, P2P is a serverless model where each computer in the network acts a node that provides the functionality of both the client and the server, transmitting data between every other node on the network. Therefore, each peer requires *O(n-1)* connections which means that client requirements increase for each new peer in the network. [9, ch. 6.]

However, P2P has been used successfully in the past with much slower network connections. This thesis does not deal with P2P any further, but it can be a valid option for some games.

2.5 Bandwidth

State updates are sent from the server in batches commonly abbreviated as *ticks* in order to reduce bandwidth**.** Thus, the *tick rate* (update rate per second) of the server determines how often the client is updated. Operating a higher tick rate requires more CPU and bandwidth for both the server and the client, but also results in a more up to date simulation, which can be critical for fast paced games. [12.]

When player input is unpredictable and synchronization is critical to the gameplay, servers usually operate on a higher tick rate and serve less players. Games which need to support more players must run on a lower tick rate to reduce bandwidth or use techniques to split the player base into more manageable chunks. For example, modern MMOs may support up to thousands of players, which is alleviated by *instancing* players into multiple servers, so that a single instance may only have up to 100 players.

Another point of optimization is minimizing *data redundancy*, by only sending the change (delta) of the previous state. Or instead of sending state, the client only sends the input commands it executes. However, since each client is performing an independent simulation, it is important to maintain synchronization with the server. [9, ch. 1.]

For larger batches of data, lossless compression algorithms can be used to reduce the message size. This is beneficial because the network I/O and download speed is more likely to bottleneck the client than running a decompression algorithm on a string loaded in memory. Otherwise, sharing code between the server and the client can greatly reduce the amount of data that is sent, by *object replication***.** For the client to spawn a new player, the server only needs to send the parameters (id, coordinates) required to instantiate the object. [9, ch. 5.]

2.6 Latency

In the earlier days, many of the initial networked games were not designed to function over long geographical distances. Games were played on a *Local Area Network* (LAN) where the network latency was not a point consideration. [9, ch. 1.]

However, on a *Wide Area Network* (WAN) latency becomes a major consideration, as more hops result in more *processing delay* (packet switching), *queuing delay* (router buffering) and most importantly *propagation delay* (physical delay) because data cannot travel faster than the speed of light. Therefore, latency can be estimated based on geographical distance, but due to the unpredictable nature of packet switching, routes may not be optimal. [9, ch. 7.]

Player inputs must always be sent to the server before receiving a response, which means that the client receives the updated state only after *round trip time* (RTT) of the message, as illustrated in Figure 4.

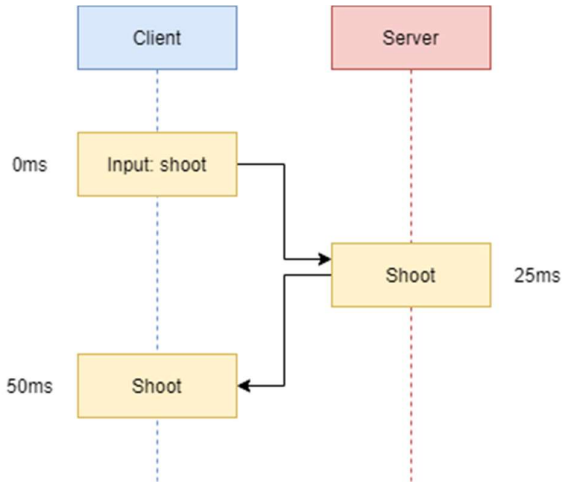

Figure 4. Input delay

As depicted in Figure 4, the RTT between the client and the server is 50 ms. Because inputs take RTT to be sent back to the client, input would feel very unresponsive especially at higher latencies. As shown in the example, the client receives the shoot command after approximately half an RTT delay since it was executed on the server.

The *client prediction* method is used to minimize the sluggishness. *Dead reckoning* is a widely used method to achieve this goal. Dead reckoning is defined as any process to deduce the approximate current position of an object based on past information [10, ch. 2.4]. Upon receiving the command to shoot, the client fast forwards the simulation by 25

ms to catch up with the server. This works very well for deterministic entities, such as projectiles.

For smooth player movement, the client shares the same code as the server for handling input so that the player can move immediately instead of waiting for a response. As a result, the local player moves immediately when an input is received as if the local player was playing a single player game. After receiving an update, the player location is corrected if necessary, which should not be the case very often in a deterministic game.

However, as a result, the client is now ahead of the server for the local player. To alleviate this problem, the server may use dead reckoning to fast forward player inputs. For better accuracy, adding some initial acceleration to player movement gives the server more time to converge with the players' location [11].

Because the server only sends state updates based on the tick rate (which is usually much lower than the frame rate), the movement of other players would appear choppy. Instead of moving the player to the received location instantaneously, the player is moved over time, by moving it through intermediary points over a duration of time which is the process of *interpolation*. However, the process adds even more latency to the client, as depicted in Figure 5.

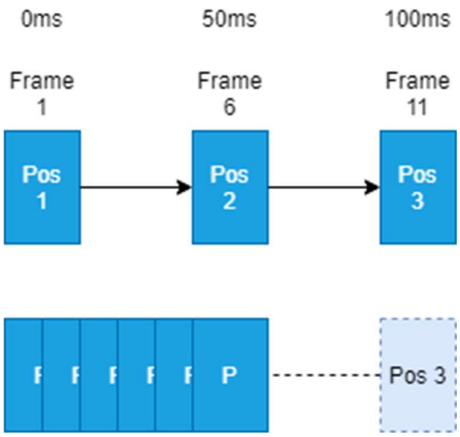

Figure 5. client interpolation

As shown in Figure 5, the interpolation delay should match the tick rate of the server so that the interpolation finishes on time for the next update, which in this figure is at 50 ms. Thus, the locations of other players are delayed by *RTT / 2* + *tick rate* without the use of dead reckoning. Therefore, lower server tick rates affect the amount of latency by increasing the amount of interpolation delay.

Servers often implement *latency compensation* to rewind time in order to see what the client saw when the command was sent. To alleviate desynchronization, by saving a short history of player positions, the server can replicate what the client saw when the shoot command was issued.

The rewind method in the Source engine is described by Valve in the following way:

> Then the server moves all other players - *only* players - back to where they were at the command execution time. The user command is executed and the hit is detected correctly. After the user command has been processed, the players revert to their original positions. [12.]

For example, if a player shoots a bullet at 10.5 client time, and the server receives the message at 10.6, the server replicates what the player saw at 10.5 client time by rewinding the locations of other players to 10.5 minus the interpolation delay. This is possible by the server always keeping a brief history of previous player locations. [12.]

2.7 Protocol

In order to understand each other, the client and server both share the same protocol. A protocol is a predetermined set of rules for communication. In the transport layer of the Internet Protocol, Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) are commonly used for packet delivery. TCP is used for connected, reliable and ordered messages, while UDP is used for sending plain connectionless messages. [13.]

In order to accomplish true full-duplex communication the communication protocol must receive some verification that the message was received. This slows down the communication, especially when the server is far away. On a web server, this transmission delay is not of importance and what matters most is that the response arrives in one piece.

TCP may be a valid choice for a game in which all data must arrive safely and in which the latency does not matter (i.e. in turn-based games) [9, ch. 7]. However, TCP carries a multitude of problems for real time games:

- Due to the ordered nature of TCP, low priority data may prevent high priority data from arriving on time. [13]
- Resending or waiting for data that is already out of date results in unusable packets using up bandwidth. [13]
- Buffering of data (Nagle's algorithm, which may be disabled) may cause the data to be sent late. [13]
- Most functionality is managed by the Operating System (OS) which makes it difficult to customize for different delivery requirements. [9, ch. 6]

It would seem reasonable to use both protocols at the same time, for example TCP for chat messages and UDP for player movement. However, both TCP and UDP are built on top of IP, which means the underlying packets sent by each protocol will affect each other, and as a result TCP tends to induce packet loss in UDP packets. [13.]

Thus, only UDP is used. Because UDP is effectively a blank messaging system that makes no guarantees that the data arrives, any desirable features of TCP must be implemented on top of UDP, depending on the type of data:

- Messages not arriving to the destination.
- Messages arriving in order.
- Flow control to limit the amount of data that is sent each second.
- Message buffering to optimize packet size for delivery.

Not all game updates are created equal. Thus, the application layer protocol may implement different options. Updates such as player movement may be sent as plain UDP, whereas it is very important for chat messages to arrive safely in order.

The protocol must also be able to verify that the message arrives without corruption. The general strategy for this is using a *checksum* algorithm that verifies that data arrives in one piece. UDP provides an automatic 16-bit checksum (optional in IPv4 and mandatory in IPv6) in the network layer so that incomplete packets are automatically dropped. Because the checksum is limited to 16 bits, statistically that means that one out of 65,536 corrupt packets are false positives, which means that it cannot be trusted for perfect reliance. If perfect reliability is a requirement, a custom checksum must be added. [13.]

2.8 Packet loss and duplication

When it comes to UDP, it is not a possibility but an eventuality that some messages will be lost or duplicated [13]. The application layer protocol must be able to track and handle packet loss.

When a server sends changes to the state, the client must have the previous state to be able to apply the changes. In case of packet loss, the player could lose a snapshot and become *desynchronized* from the servers' state. There are some solutions to this problem.

- The server tracks the latest snapshot each client received to calculate a delta snapshot (differences to previous state) for each individual client.
- The server tracks the state for each client and resends a complete snapshot of the world in cases of desynchronization.

If the model is not server-authoritative, the solution for movement can be simple. Because the player movement can be simulated on the client, the client only needs to update the player coordinates to the server. If packets are lost, the player stops moving on the server, and upon receiving the latest packet, the player location is corrected.

For a server-authoritative model, input should include a *sequence number* (borrowed from TCP) so that the server can track which inputs are missing [15]. The use of these headers is shown in Figure 6.

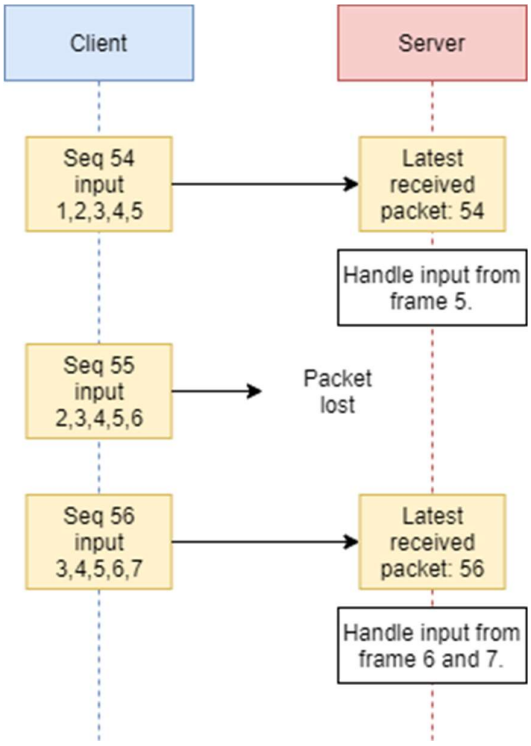

Figure 6. Packet loss for client input [15]

As illustrated in Figure 6, the server utilizes a message sequence number to handle incoming input. Each message contains a history of inputs from the previous five frames in case of lost packets. Thus, the server extracts the missing inputs from the message which reduces the latency requirement of requesting missing inputs. [15.]

If no inputs are arriving, the server can assume that the previous inputs continue (player keeps moving) or assume nothing at all. Using the sequence number method allows the server to replay the inputs when they finally arrive to correct the game state. [15.]

However, including a history of inputs adds redundant size into the messages, which is the tradeoff for less latency when packets are lost. While this works for inputs which are relatively small in terms of bytes, for other cases the protocol must be able to request missing packets. An example scenario is illustrated in Figure 7.

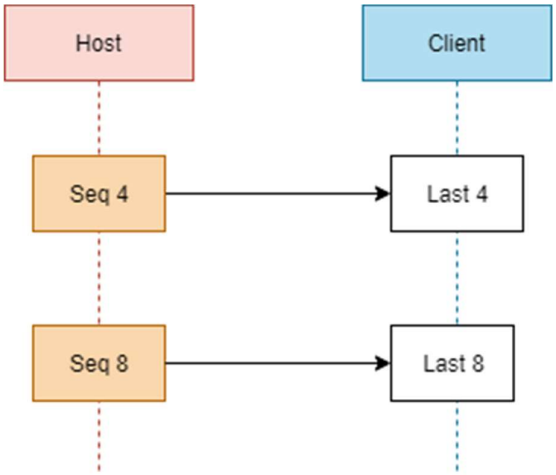

Figure 7. Multiple missing packets [15]

Given the situation in Figure 7, the client needs to be able to tell the host which packets are missing. Therefore, the *acknowledgment bitmask* and *last received* headers are used to report missing packets, as illustrated in Figure 8. [15.]

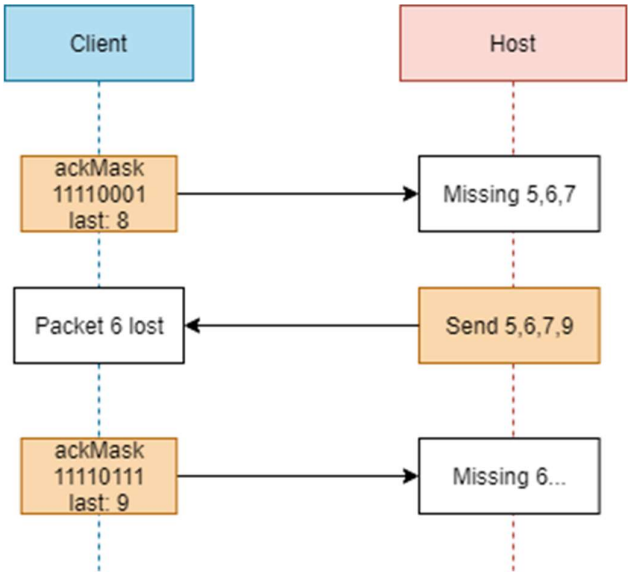

Figure 8. Using a bitmask for packet loss redundancy [15]

As illustrated in Figure 8, the ackMask header is a sliding window of previous packets, where each bit corresponds to a previous message. Upon parsing the header, the corresponding sequence number is calculated so that the missing message can be retrieved from history and resent. [15.]

2.9 Packet fragmentation

For optimal performance, packets should fit a single *ethernet frame***,** up to 1500 bytes maximum [5, p. 1]. Packages larger than the *maximum transmission unit* (MTU) are automatically fragmented in the network layer IP headers, which results in the following issues:

- Each fragment must contain an additional set of headers [5, p. 2].
- If fragments are lost in transit, the receiving host must drop the entire packet, which results in an increased likelihood of packet loss [9, ch. 2].

However, the same problem applies to small packets as well. The minimum length of an IP header is 20 bytes [18, ch. 5], which results in a much larger percentage of headers in small messages. Thus, the optimal packet size is as large as the MTU allows. Therefore, it is recommended to implement buffering functionality that batches multiple updates into a single packet, without exceeding the limit.

2.10 Differences to web servers

All computers connected to a network are assigned an IP address, which is the address where the computer may receive packets. Most public web servers such as Google are found on a Domain Name Server (DNS) which translates requests sent to google.com to an IP address. [18, ch. 11.]

Each message sent to an IP address must include a destination port, also known as a network socket, which is an endpoint that determines how to handle the incoming message [9, ch. 3]. Web servers typically use default ports 80 for HTTP and 443 for HTTPs, which are assumed by the web browser by convention.

A message is forwarded on the network until it reaches the destination address or reaches a time out. The server responds to the request by sending the response back to the sender's (public) IP address, which is typically a router that forwards it to the computer [18, ch. 7].

A traditional web server is *stateless*. The client sends a request asking the server for a file, such as a web page when navigating to Google on a web browser. The browser sends an HTTP request to the server, which is a relatively lightweight protocol designed for infrequent requests from a high number of clients. The requests are ad hoc in nature, meaning that the server does not keep track of client state. A web server may protect resource endpoints by requiring an *authentication header* which identifies the client sending the HTTP request. [19.]

For a video game, the server needs to be *stateful*, which means it must track the state of each client [19]. For the initial connection, the client must send a connection request to the server (address and port), which accepts the request after a virtual handshake and adds it to a list of active clients. After this initial connection the client and server may communicate between each other by a constant stream of messages. If the stream stops abruptly, the process may be timed out and consequently disconnected automatically.

In addition, user hosted game servers often run into a problem with *Network Address Translation* (NAT). When the server is running on a private network, the router needs to be configured to forward incoming messages to the host. The problem of this solution is that it requires technical knowledge from the users. Therefore, the *STUN* (see Appendix 1) protocol is an option to be considered. [9, ch. 2.]

3 Methods and material

3.1 LÖVE2D

LÖVE is an open source 2D game framework and does not provide a graphical user interface and tooling that many other development platforms such as Unity provide. The framework utilizes *Lua* as the programming language. The benefit of using LÖVE is that is not being tied to a development platform's ecosystem which abstracts the lower level. Typically, this results in more work for more control as development platforms are designed to increase productivity. [1.]

LÖVE may be used to build games on all platforms, but for the Thesis Windows 64-bit was chosen. The difference in platform is not significant, provided that the project source code does not depend on platform specific binaries.

The core functionality of LÖVE can be summarized in three main callbacks. The *load* callback handles the one-time setup, such as loading all the files, assets and graphics settings. After the setup, LÖVE starts the game loop, calling the *update* callback passing delta time every frame. For example, handling the simulation of physics, such as moving the player character. [1, ch. 4.]

After the state of the game has been updated and the player has been moved, LÖVE clears the graphics canvas which removes an image of the player in the previous location. LÖVE then calls the *draw* function, which is where the developer places things they want to draw onto the canvas, such as the player with new coordinates. [1, ch. 4.]

Other than the main callbacks, LÖVE provides an extensive API for user input, audio, images and file system [1, ch. 4]. Other common features such as networking and physics engines are provided by open source libraries.

3.2 Lua

Lua is a light-weight extension language that is commonly used in game engines. It was written in clean C with a small memory footprint which makes it a good fit for game engines. Most game engines are written in C++ and Lua can call functions written in C. [2.]

All values are first-class values, which means functions can be stored in variables, passed as arguments and returned as results [2]. Lua's syntax resembles that of Python. However, the language is easy to pick up with experience in any scripting language. The most notable differences are:

- The only available data structure is a table (*local table = {}*), which has the functionality of a Java hash map or a Python dictionary. Values are indexed by anything other than a nil [3, ch. 2.5].

- Tables can be customized to create other data structures or objects. This can be expanded by customizing *metatables***,** which controls how the table behaves for different operations. For example, the addition operation can be customized for a table variable. [3, ch. 13.]

- All numbers are double-precision floating-point numbers [3, ch. 2.3].

LuaSocket was the library of choice for networking. It composes of a C core that supports TCP and UDP and Lua modules for common protocols. Sending and receiving messages are automatically encoded and decoded as strings, which means that it is not possible to send raw bytes. [4.]

3.3 Programming patterns

Designing a coherent architecture for a game is a difficult task, due to the amount of different unrelated components which are dependent on each other. When a player jumps, the physics component calculates the player's velocity and moves the player, the

audio component plays a jumping sound, and the rendering component updates the animation. [7.]

For a complex architecture it is important to avoid *tight coupling* of components. Each component should be easy to understand without prior knowledge of its dependencies. In practice, one common mistake is the overuse of inheritance, which leads into a situation where it becomes unclear what the class does without reviewing what each parent does first. Instead, it is a growing trend to favor *composition over inheritance* [7, ch. 5]. Thus, a primary design goal was to decouple the game client (rendering) from the networking client.

Bob Nystrom provides a concise description in the book *Game Programming Patterns*:

> You can define “decoupling” a bunch of ways, but I think if two pieces of code are coupled, it means you can’t understand one without understanding the other. If you de-couple them, you can reason about either side in-dependently. That’s great because if only one of those pieces is relevant to your problem, you just need to load it into your monkey brain and not the other half too. [7, ch. 1.]

When it comes to networking, an *event-driven architecture* handles communication as a one directional stream, which includes the benefit of not having to wait for a response. The concept is illustrated in Figure 9.

Figure 9. Event driven data flow.

As showcased in Figure 9, the client receives state updates as a continuous stream from the server without explicitly requesting for them. On the contrary, in a *request-response* architecture the client must send a request to receive an update.

Programming patterns such as the *Observer* are used to achieve the same result for software components. For example, the physics component for players handles collisions. Upon colliding with the ground, the player needs to play a sound effect depending on the type of collision. [7, ch. 2.]

However, Nystrom also reminds that the pattern should not be overused:

> The observer pattern is a great way to let those mostly unrelated lumps talk to each other without them merging into one big lump. It's less useful within a single lump of code dedicated to one feature or aspect. [7, ch. 2.]

Instead of adding the sound effect to the physics component, the collision function notifies all *listeners* of a player collision. A *callback* is a function that is to be executed after an event, which in this example would be the player collision [20]. The collision type may be passed as a parameter, which means different listeners such as the audio engine can play a different sound depending on the type of the collision.

The code is easier to understand, because the classes are now their own complete entities. The physics class can be used for things other than audio as well, granting much more flexibility in the future. However, this type of pattern should not be used when two components are intrinsically dependent on each other because it also makes the link between the components more abstract.

4 Architecture

4.1 Goals of the Thesis

The server should be able to handle ten players. This depends not only on the hardware running it, but mostly importantly the quality of the internet connection and the amount of bandwidth available.

The game server can also be hosted on a personal computer which is running the game client, which allows players to host and play simultaneously. This is how single player video games which need multiplayer support are often designed, meaning that the single player version is always hosting a server in local play.

The network architecture must fulfill the following criteria:

- The communication protocol minimizes overhead for constant updates such as player movement.
- The client must be able to recover from packet loss.
- The client needs to keep track of current RTT and attempt to reconnect if no response is heard from the server for a long enough time.
- The server must automatically disconnect clients that are not heard from, but also return them to where they left off when they reconnect.
- The server needs to be authoritative for bullet collisions.
- It should be easy to add new abilities and objects to the game state.

Perfect optimization is not the goal. The goal is to have a functioning messaging system, instead of optimizing latency compensation. It means these goals may be achieved with methods that are “good enough”, but with the understanding on how the solution could be optimized.

The game should be playable with an RTT of 50 ms. This is a very low latency but still a reasonable expectation for a multiplayer server in close geographical proximity. Higher

latencies may feel unfair due to lack of latency compensation and server-authoritative bullet collisions.

4.2 Client and host classes

The client and the server share some components, which makes it much easier to use the same programming language and libraries. The following features can be shared:

- Message format, parsing and compression.
- Game state update loop, which continues to run even if messages are lost.
- Tick rate timer for fixed timestep updates.
- Checking for connection timeout.

Thus, it is reasonable for the client and host to share a parent class which abstracts the same update flow for both entities. This is further detailed in Figure 10 and Figure 11.

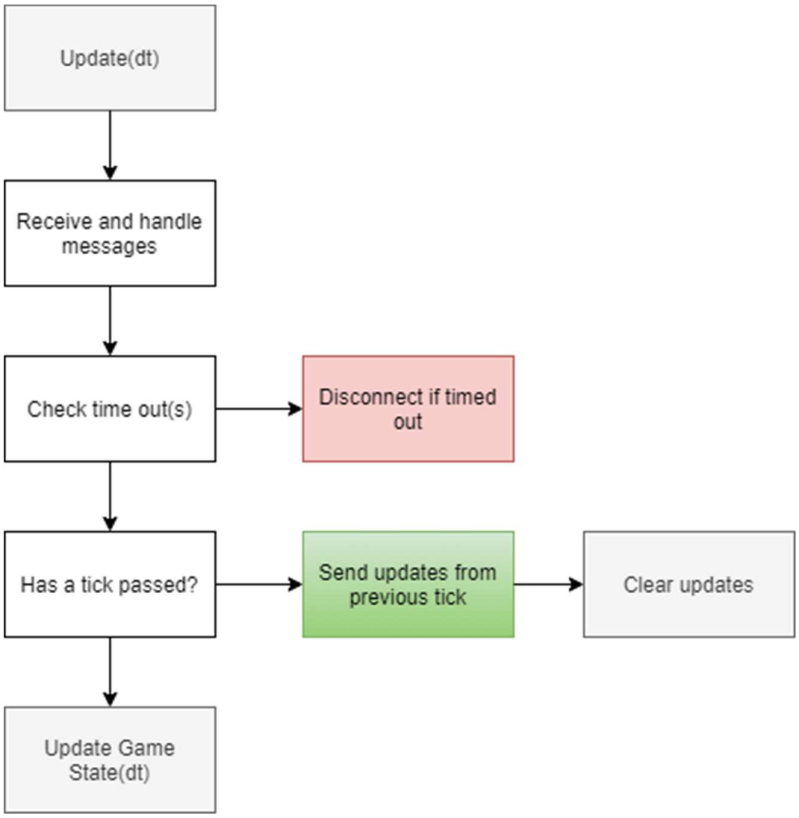

Figure 10. Connection update loop

As shown in Figure 10, the loop receives and handles messages on every frame, and updates timers based on delta time. Thus, the loop can be run directly from LÖVE's update function. A dedicated server calling this loop is illustrated in Listing 1.

```
while running do
    currentTime = host.socket.gettime()
    local dt = currentTime - previousTime
    previousTime = currentTime

    host:update(dt)
end
```

Listing 1. Calculating delta time for the update loop

As demonstrated in Listing 1, the update loop can be run as a continuous loop which calculates delta time on every iteration. Unlike a fixed timestep (or a limited frame rate), the loop never sleeps, which reduces latency for network requests.

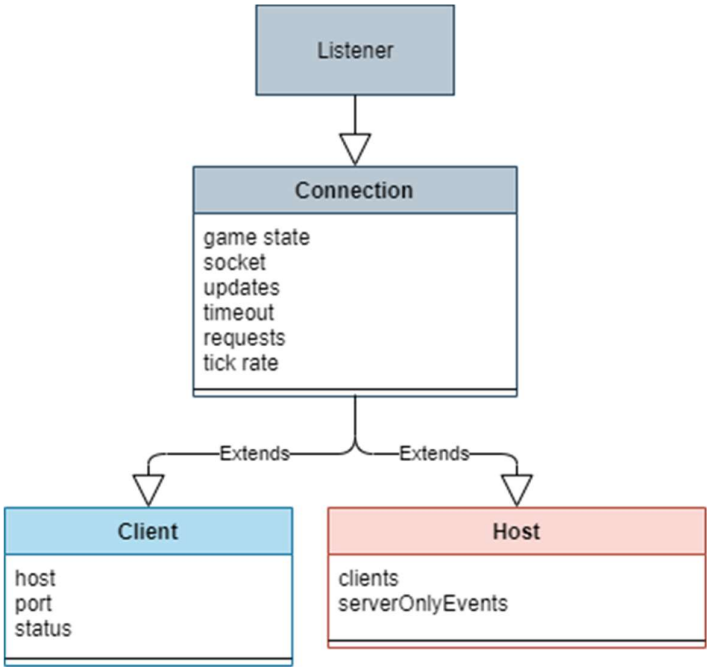

Figure 11. Client and host component diagram

As shown in Figure 11, the main difference is that the host class handles many connections and the client class only one. The host class implements additional logic in the form of a *server only* events, which listens to game state events that are server-authoritative (such as bullet collisions) and updates the clients accordingly.

The game client (not to be confused with the network client) runs the network client on a high level of the component hierarchy. This allows the game client to listen for network events, such as the connection timing out so that the scene may be changed to loading. A diagram showcasing the game in Play Scene is shown in Figure 12.

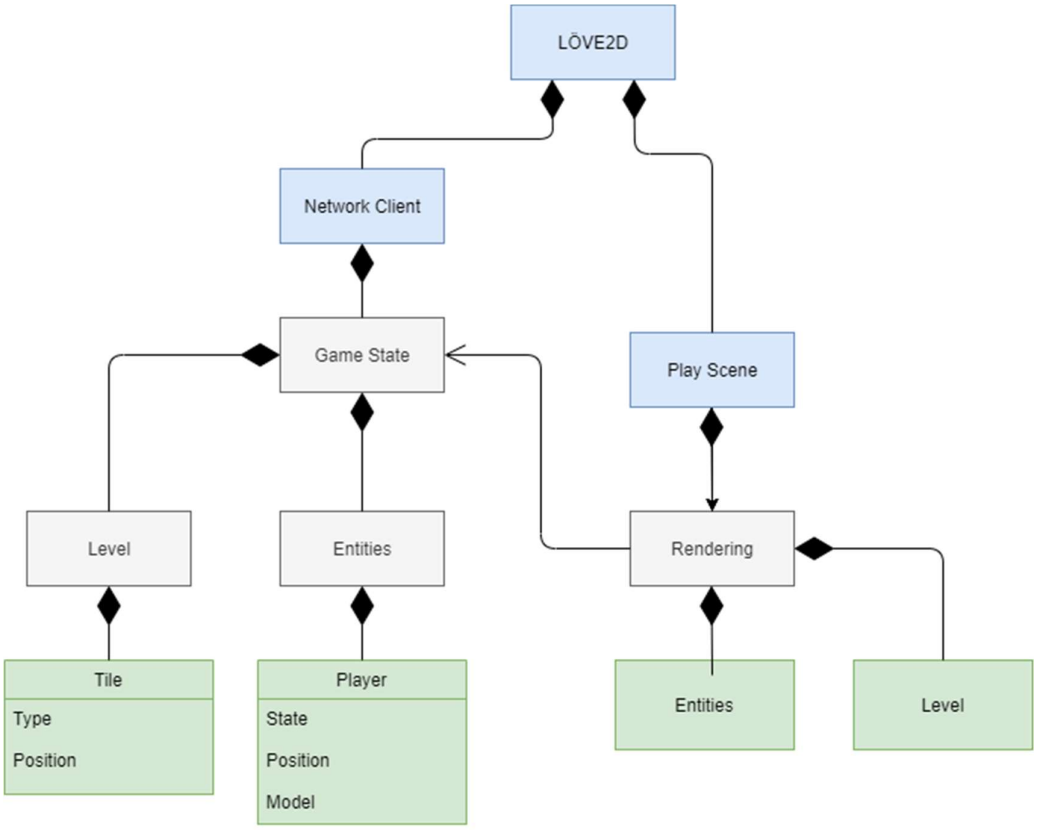

Figure 12. Game client component diagram

As shown in Figure 12, the game scene and network client are separate entities. The rendering engine could be replaced because the network client is not coupled to the rendering component.

4.3 Message handling

Most of the communication should be event driven, which means that updates are automatically sent based on the update rate. A basic request does not require a response, and thus it does not wait for one. It is simply a *Remote Procedure Call* (RPC)**.** If a response is sent, it is not explicitly linked to the request. Every message follows the same class schema, depicted in Figure 13.

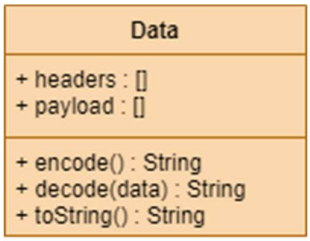

Figure 13. Data class

As shown in Figure 13, the schema allows data to be constructed as an object, which is easy to pass directly into the send function or to push into the updates table to be sent on the next tick. To *batch* multiple requests into a single message, a Data object can wrap a table of Data objects within the payload. In addition, a batch request must contain the batch header to be processed correctly.

The *encode* function returns the data as a string, which contains an optional MD5 checksum and JSON for the data object which is compressed using the LZW compression algorithm. This option is provided as an optional backup for the UDP checksum for the reasons stated in chapter 2.7. The *decode* function parses a data string and validates the checksum (if set), returning the Data object.

Both the client and the host should be able to send and receive requests. This allows the customization and addition of new requests in both ways. For each incoming message, the message headers are parsed to determine the type of request from the *request* header in order to invoke the RPC. This is illustrated in Figure 14.

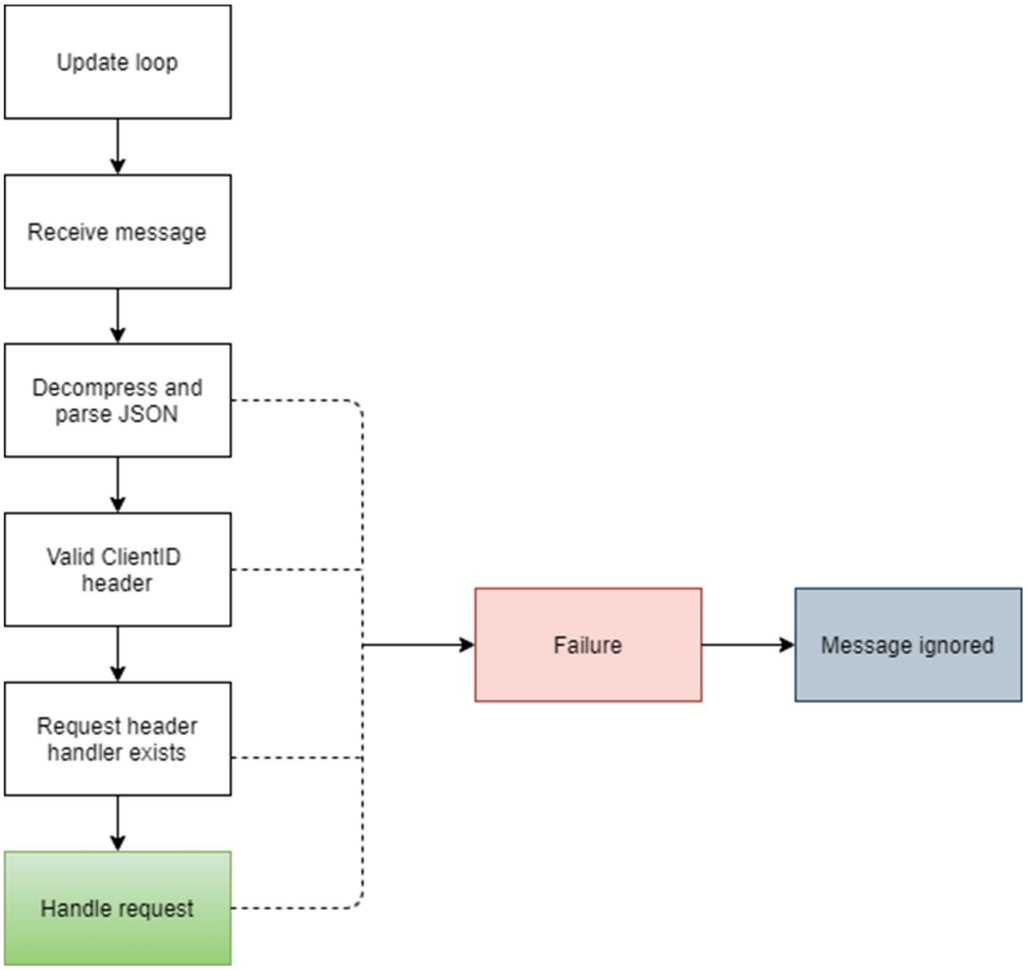

Figure 14. Message handling

As shown in Figure 14, any request (other than the connection request) must also contain a client identifier, which is then validated to match the incoming IP and port. These headers must be included in every message, or the message is dropped.

As explained in the previous chapter, the tick rate is only used to limit how often updates are sent from the update buffer. Messages or responses that need to be sent immediately, such as a player disconnecting from the server, can be sent directly without waiting for the next tick.

4.4 Latency and time out

To be able to track latency, the messages must contain a timestamp for when the request was sent and received. The request must always be initiated by the client, which upon receiving a response is able to calculate how long it took for the request to arrive and return. The request is demonstrated in Figure 15.

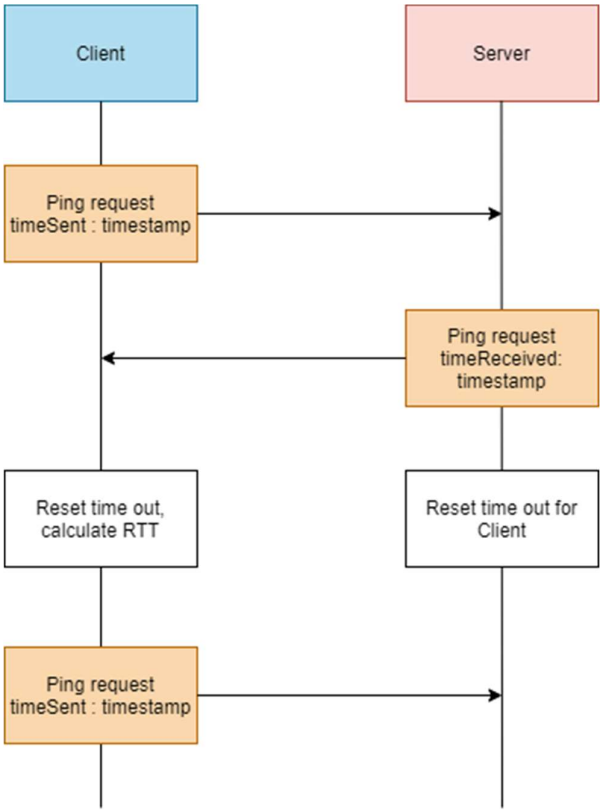

Figure 15. The ping request

As shown in Figure 15, the ping request also acts as a keepalive, refreshing the time out for both parties. This logic can be combined with periodic updates, which eliminates the need for a separate request.

4.5 Duplex messaging

World snapshots, such as sending the world tile map upon connection, require a response from the client that the data was received. The snapshots must arrive in order.

To ensure order of delivery, messages are placed into a queue. The initial message is sent immediately, and upon receiving a duplex message an acknowledgment response is sent back to the sender. After receiving the response, the next message in queue may be sent. Without response, the queue can retry automatically until a response is received or a time out is reached. An example of a duplex message is depicted in Figure 16.

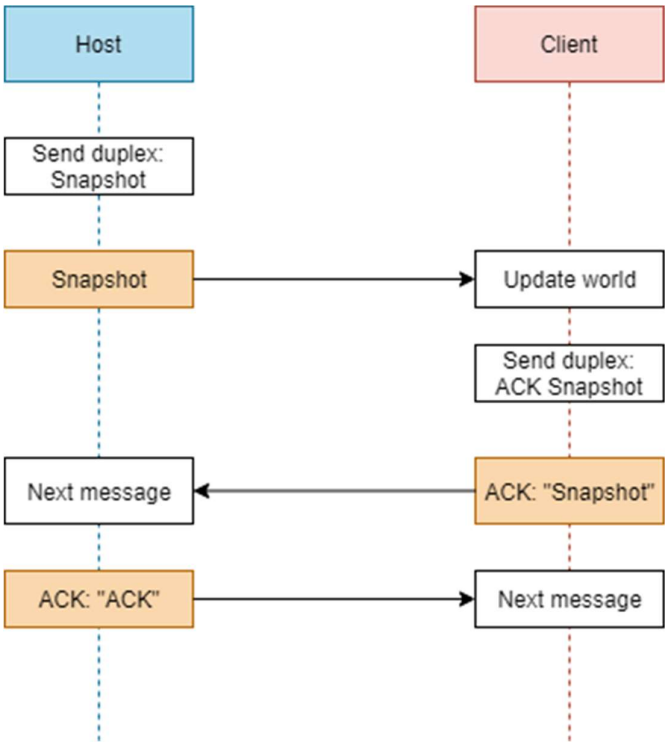

Figure 16. Example of duplex communication

As depicted in Figure 16, the ACK response is also pushed into the duplex queue, in order to handle packet loss. If the ACK response is lost, it is sent again until a response is returned.

4.6 Snapshots

The state contains all the entities which are updated in the game loop. Therefore, all states must me serializable so that a snapshot may be sent and received. Every component under *GameState* must implement a *getSnapshot* and *setSnapshot* function, which return or set the game state based on serialized data. The basic state hierarchy for the game is presented in Figure 17.

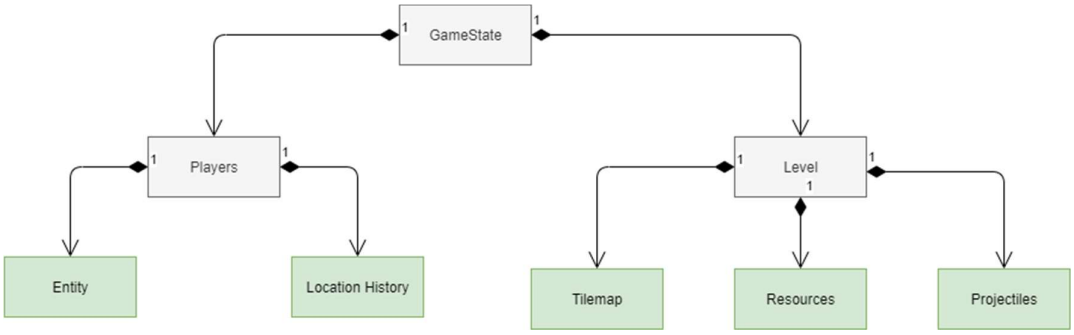

Figure 17. Game state components

As shown in Figure 17, the components of the game state may contain more components. As a result, all components in the hierarchy implement a function to read and write snapshots. Calling the *getSnapshot* function for *GameState* returns the full snapshot of the state hierarchy which may be sent over the network.

A snapshot contains a complete copy of the current state, which means that a snapshot may be very large in size. Thus snapshots must be compressed without losses. To minimize the data required for transport, the *getSnapshot* function should be able to compress the data and the *setSnapshot* function should be able to read the compressed format.

To provide an example, the tile map may contain different types of tiles. Instead of sending all the properties of a tile, only an identifier to the tile type is sent. An example is shown in Listing 2.

```
{
  {0,0,0,0,0,0},
  {g,g,g,g,g,g}
}
```

Listing 2. Serialized tile map

In Listing 2, empty tiles are set as zeroes, which allows array indices to be translated directly into game world coordinates. If tile width and height is 20, the second element of the second table has the world coordinates *(20, 20)*.

Another point of optimization, which is more specific to this type of game, is chunking. The idea is based on a common rendering principle, which is to not update what the player cannot see. The world may be very large, and the player does not need to load the complete map upon connecting to the server. Instead, the player receives snapshots of the visible chunk the player is located in. Thus, the server only needs to send the current chunk to the player, shown in Listing 3.

```
{
  { x: 720, y: 720 },
  {
    {0,0,0,0,0,0},
      ...
  }
}
```

Listing 3. Serialized chunk

In Listing 3, the starting coordinates are included to translate tiles from the array to world coordinates. Thus, *x* and *y* are added for each tile, meaning that the first tile in this example has the world coordinates *(720, 720)*.

However, this also means that the client must take resolution into account. To keep it simple, any resolution in the game client scales the camera so that the *field of view* is constant on all resolutions. If one tile is *20 * 20* pixels, the screen contains 2304 tiles on a resolution of *1280 * 720*. Therefore, a large game map contains up to hundreds of thousands of tiles. The minimum chunk size is illustrated in Figure 18.

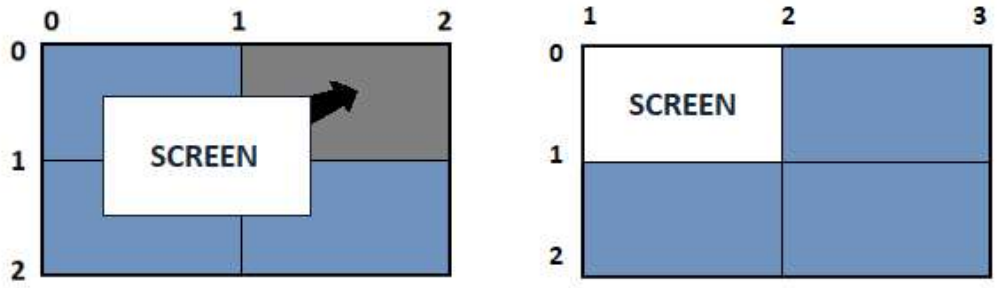

Figure 18. Player changing chunks

As shown in Figure 18, to ensure all assets within the current chunk are loaded, the height and width of the chunk must be double of the player's maximum field of view. Therefore, the chunk consists of four screen-sized segments, which are updated. When the screen no longer collides with the current segments, a new chunk is loaded.

Player movement is tracked within a location history table. Each time a player moves into a different chunk, an event is triggered on the server. The server checks the player's location history, to see if the player should receive a snapshot of the new location.

4.7 Events

In order to batch updates for every tick, an update buffer needs to be implemented. On every tick, the updates are sent over the network and moved to a temporary history sequence in case the updates must be resent. The update buffer contains:

- Player state and movement vectors.
- Events such as using abilities or changes in the tile map.

Player movement updates can be sent as pure UDP messages. If a packet is lost, the player will simply not move on the client. Because the data is time critical there is no benefit to resend lost packets. However, this also means that the player data must always be a full snapshot of player movement.

Events, such as using an ability must have reliable but not necessarily strict order of delivery. Thus, the event protocol must be able to track packet loss and automatically resend packets that have not arrived. The events protocol handles packet loss as described in chapter 2.8, depicted in Figure 19.

Figure 19. Events message headers

As it can be seen in Figure 19 each event contains to handle packet loss and calculate latency. Because each event packet corresponds to a single tick, it is sent as a batch of 0 or many requests.

4.8 Abilities

An ability is a function the player entity can execute in the direction or position of the cursor. This means that to execute an ability, it requires access to the player entity, position, and cursor coordinates. To give an example, an ability can mean destroying tiles or shooting bullets. An example of an ability request is illustrated in Figure 20.

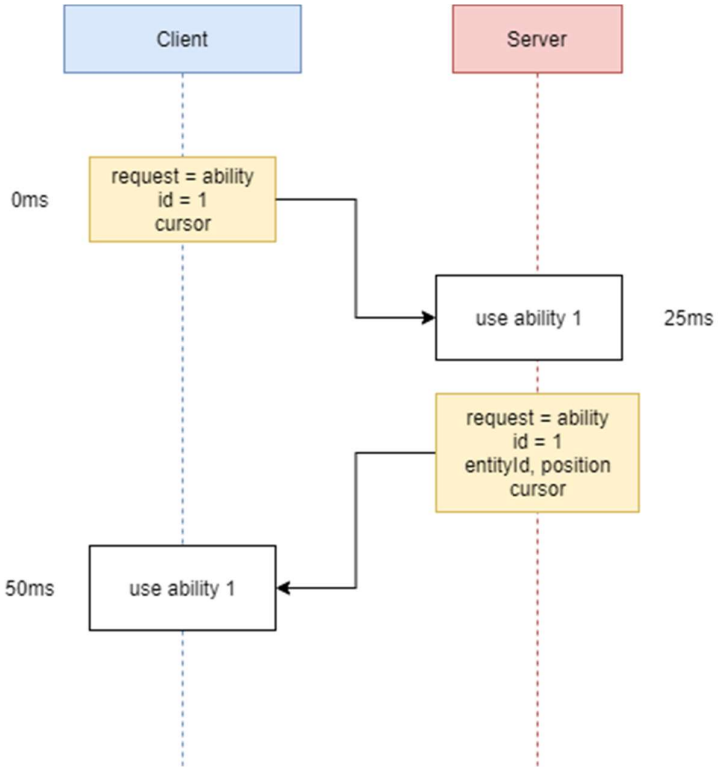

Figure 20. Ability request

As shown in the example in Figure 20, after receiving user input to use an ability, the client sends a request to the server passing an ability identifier and cursor coordinates within the data payload. The server first validates and executes the ability, and then sends a response to all clients that an ability was triggered by a player in a position and cursor coordinates during the time of execution. This allows all the clients to then replicate the ability with the same state the server used.

Ability execution logic is found directly under game state, which is shared by both the client and the server. Therefore, adding new abilities into this system is very easy.

5 Implementation

5.1 Starting point

The networking architecture was to be implemented on top of a single player prototype. Yet, it became clear that the prototype would be very difficult to convert into a multiplayer. The prototype was developed without the use of any libraries, which is a great learning experience, but this added some unnecessary difficulty on top of an already challenging project. A screenshot of the prototype is shown in Figure 21.

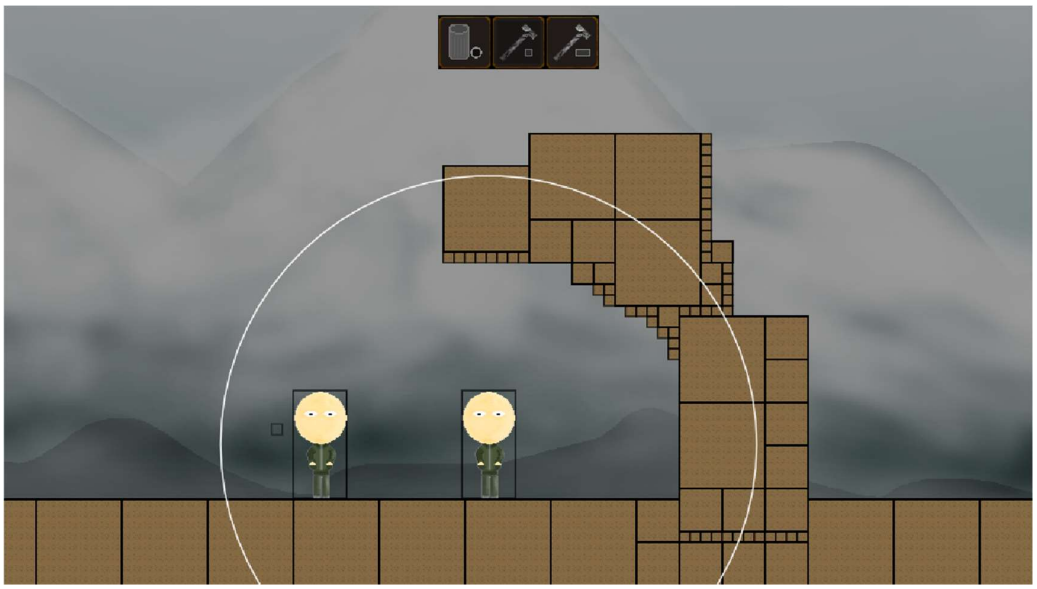

Figure 21. Single player prototype

As shown in Figure 21, the prototype included a recursive algorithm which automatically combined tiles. It was removed due to code complexity and possibility of stack overflow. The solutions to the core elements of gameplay such as physics had flaws, and fixing issues took most of the time there was to carry out the project.

It is not necessary to reinvent the wheel if an effective solution already exists. Libraries are designed for the sole purpose of solving a problem, which usually results in better optimization and features.

The project was restarted, adding the components that were removed one by one when the need arose. Collision and user interface code was replaced with open source libraries that provided better solutions. Even though most of the prototype's code was deleted, it did not mean the previous work was a waste of time. Though it may have felt bad to remove features that took a great deal of work, the real value came from what was learned.

It was necessary to start small and to get a grasp of the easier networking concepts before moving onto more difficult ones such as latency compensation, optimization and handling of packet loss. As shown in Figure 22, it is often a good idea to make a throw-away prototype to test an idea or a library before embedding it into a larger project.

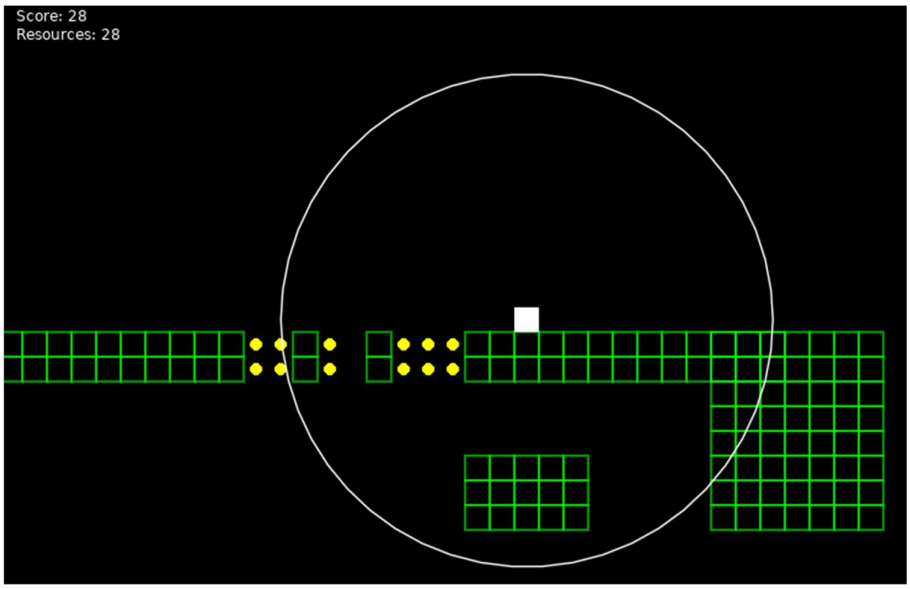

Figure 22. Physics prototype

The prototype in Figure 22 was used to test an open source physics library for the game's mechanics. Without any thought for structure, the prototype could be built in a matter of two hours to test core mechanics in a single player environment.

Even though a clean, loosely coupled architecture is the end goal, it does not have to start there. Premature optimization can lead into no progress being done at all, so a

balance must be struck. To keep it simple, the goal was to leave the codebase better with each commit than it was before. Review the changes until that level is reached.

5.2 Initial networking

LÖVE's networking tutorial was used as an example to set up an initial communication loop between the client and the server. It is an infinite loop that receives UDP messages via the LuaSocket library and parses them for updates. An example of parsing is shown in Listing 4.

```
entity, cmd, parms = data:match("^(%S*) (%S*) (.*)")
```

Listing 4. Parsing messages with regex

As depicted in Listing 4, the message was parsed using *regex* to split the request type and parameters from the string. After adding more request types, it became abundantly clear that parsing strings with regex would be difficult, having to customize the match pattern for each different request. As Jamie Zawinski famously articulated in 1997, "Some people, when confronted with a problem, think 'I know, I'll use regular expressions.' Now they have two problems."

Thus, the idea of using regex to parse the payload was scrapped. Option one was to send Lua code over the UDP connection and try to execute it on the receiving end. This would also not be a good solution because of the possibility of executing malicious code. The data should be serialized in a format that would not be executable, common options being *JSON* or *XML*. JSON was chosen due to being less verbose of the two.

The Client and Host classes then were implemented using the shared parent class as described in chapters 4.2 and 4.3. An example code listing of shared update loop and message parsing is included as Appendix 2.

5.3 Connecting to the server

To be able to connect to the server, the client returns a *coroutine* that can be used to retry sending connection requests without blocking the game loop. Coroutines can be compared to threads, in that they yield control back to the caller, but need to be restarted manually to continue execution. After a coroutine finishes, it is dead, and cannot be re-started again. The connection coroutine is shown in Listing 5.

```
function Client:connect()
    self.status.connecting = true

    return coroutine.create(function()
        while self.status.connecting do
            self:send(Data{request = 'connect'})
            coroutine.yield()
        end
    end)
end
```

Listing 5. Connection request coroutine

The connection coroutine in Listing 5 is called until a response is received or a time out is reached. Now the client can send a request to connect to the game, which returns a snapshot of the game state and an entity identified by the client *identifier* to control. The server uses the client identifier to maintain the connection.

The client identifier is henceforth required in all requests to the server as demonstrated in Listing 6.

```
function Host:validRequest(data, ip, port)
     if data.headers.clientId then
          local client = self.clients[data.headers.clientId]

          return client and client.ip == ip and client.port == port
     end

     return data.headers.request == "connect"
end
```

Listing 6. Message validation

As shown in Listing 6, the host validates that all requests other than a connection request include the identifier header, and that the message is sent from the corresponding IP and port of the client.

5.4 Player movement

After connecting and receiving the player entity, the client takes control of the player entity, and starts sending updates to the server based on the clients' tick rate to the server, as demonstrated in Listing 7.

```
function Client:sendUpdates()
    if self.status.connected then
        self:send(self.updates:getEntities())
        self:send(self.updates:getEvents())
        self.updates:nextTick()
    end
end
```

Listing 7. Client tick handler

The *sendUpdates* function in Listing 7 is called from the update loop after a tick has passed. The client sends updates of entity state (location, movement vectors) to the server on every tick, which means that player movement is client authoritative.

Both the client and the host update the game state for physics. Thus, if packets are lost the players are still affected by gravity and movement vectors. This solution was only chosen for its simplicity because cheat prevention was not a primary goal of this project.

At this stage, the player could connect to the server, receive an entity and continuously update the player location. However, other players would only update once every tick, making movement appear very choppy.

Interpolation can be implemented by using a library that provides a function for *tweening*. The function is commonly used for other areas of game development as well, such as animations. Usage is demonstrated in Listing 8.

```
function Entity:tween(time, state)
    Timer.tween(time, {
        [self] = { x = state.x, y = state.y}
    })
end
```

Listing 8. Entity interpolation

As shown in Listing 8, the tween function moves the coordinates of the entity to the target coordinates over a time period. The function is called by the client upon receiving movement updates for other entities from the server. The time parameter is adjusted based on the tick rate of the server, as recommended in chapter 2.6.

5.5 Level snapshots

Upon connection, the player needs to receive the state of the world from the server. Thus, the world tile map was to be serialized as described in chapter 4.6. At this point in the project, it became clear that even a compressed tile map results in a large packet, which is why the chunking method was designed to alleviate the problem. After implementing chunking, the number of tiles in the world could be multiplied without any noticeable effect on client performance.

However, even with the chunking method, sending full snapshots of the tile map caused the client to jitter when loading a new chunk. To reduce the amount of jitter, the player location history also had to track the previous time the player visited the chunk, so that a player moving between the border of two chunks multiple times would not have to load a new snapshot every time.

In order to combat desynchronization, as a basic a *failsafe* method the server would resend a full snapshot of the current chunk for each client every 30 seconds. This was a simple yet inefficient method to synchronize all clients, because all clients would receive the update even if they were synchronized. A more optimized solution would track the game state of each client, and only send a full snapshot in cases of desynchronization.

5.6 Abilities

In addition to networking, the ability system required an interface to select an active ability. Therefore, the client required a graphical interface and a hotkey system. The toolbar could handle a maximum of ten abilities, each mapped to a hotkey which could be set by the player. Clicking on an ability on the graphical interface simulates pressing the assigned key, which triggers the ability.

Two abilities were implemented. The first one destroys tiles in a rectangular area and can only be used in proximity of the player. If the player cursor is too far away from the player at the point of execution, the ability is not executed. Destroyed tiles leave resources that can be gathered by walking over them. By colliding with these resources (server-authoritative), the player gains resources that may be used to shoot a bullet towards the cursor position. Figure 23 showcases a screenshot of the game after the implementation of the ability system.

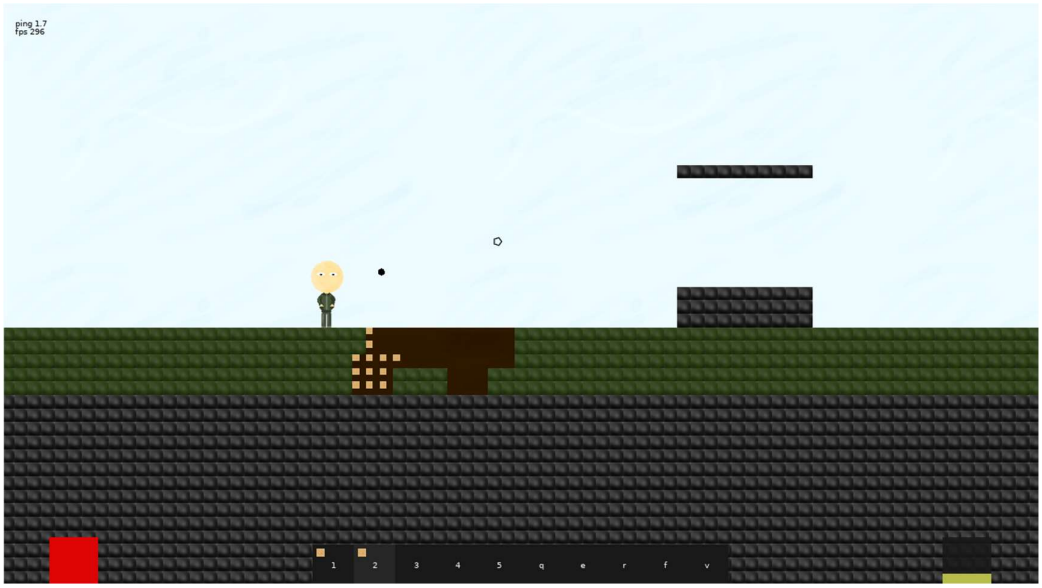

Figure 23. Shooting a projectile

In Figure 23, the second ability (shoot) is active. This changes the cursor to a circle and clicking the left mouse button sends out the ability request with mouse coordinates to be

played on the server. The server executes the shoot command, using up one resource to shoot a bullet and sends a request for the client to replicate the ability.

Latency compensation was not implemented. Thus, a problem arises with projectiles. When a player shoots a bullet, it ends up being behind the server which causes a mismatch in location. A possible fix would be to use the dead reckoning technique. First, it is necessary to calculate when the bullet was spawned on the server (based on the *sentTime* header) and then the bullet is accelerated until it catches up with the server state.

6 Problems

6.1 Movement

Simulating the movement on a slower tick rate on the server caused some unexpected problems. The server, only updating the game state 20 times per second made the player fall through the ground. The collision detection logic for players did not check for intermediary collisions upon moving the player to the next calculated position, which was a much greater distance on a lower update rate. The solution to the problem is illustrated in Figure 24.

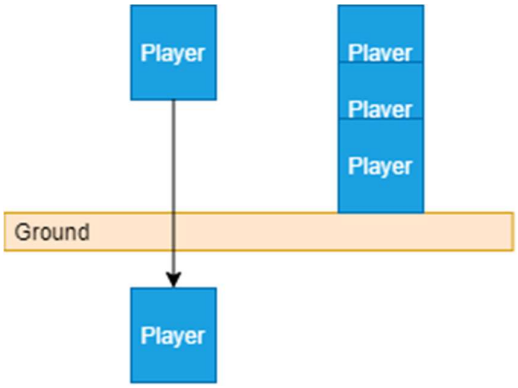

Figure 24. Collision fix

As shown in Figure 24, the solution was to move the player incrementally instead of teleporting him/her to the next calculated location. This is how most physics libraries are designed, which is why the project adopted an open source library, *bump.lua,* to handle rectangle collisions.

Even the most basic rectangle collision detection fails if the collision margin is very small. For example, if a bullet is moving 100 pixels on every frame, the bullet may skip over objects if the intermediary points are not checked. However, to achieve perfect collision the bullet would need to be moved pixel by pixel for every frame. This would result in 99 additional checks for each bullet, which would come at a great performance cost.

6.2 Inheritance

Because both the Client and the Host classes share the same protocol, they were designed to share an abstract parent class which provides a common interface for parsing requests. The difference was that the client only communicates with the server, whereas the host communicates with many clients.

However, even this small amount of inheritance proved troublesome at times because Lua as a language provides little support for Object Oriented hierarchy. Unlike in a language such as Java, it is not as easy to validate that child classes must implement the declared abstract functions, which the parent logic has relied on.

To provide an example, the update loop would trigger the *sendUpdates* function (see Listing 7) every time a tick had passed. For the client, it would retrieve the event buffer, send it to the host and reset the buffer for the next tick. The host shared the functionality, looping the function for each client. As a result, everything worked when only one client was connected to the server.

However, after a second client connected, the loop would clear the events on each iteration of the loop, sending the second client an empty update packet, which meant that for each client *n,* the client was *n - 1* packets behind. Because the event messaging protocol was able to handle packet loss, it was not obvious that there was a flaw in the system. It was discovered due to another unrelated bug, which caused desynchronized clients to crash.

As suggested in chapter 3.3, favoring composition would have been a better choice. Instead of sharing the same parent, the Client and Host classes could have shared the same logic in the form of components such as a Message Parser and Tick Rate Timer.

6.3 Snapshots

In hindsight, the duplex queue implementation was very problematic because the queue could get stuck waiting for a response. The idea was to mimic the acknowledgement

system from TCP, but it should have included sequence numbers to allow the messaging protocol to temporarily store packets in memory before receiving an acknowledgment. Therefore, the receiver can receive multiple updates without having to wait for an acknowledgment on each one. However, the solution can be used for infrequent updates that are not time critical, and thus it was good enough for full snapshots of the game state.

Even with chunking, the snapshots were large, and noticeable jitter could be experienced on the client when loading one from the server. Instead of sending full snapshots, the system should have been designed to calculate delta compressed snapshots. This would be the first point of change if the project were to be continued.

7 Conclusion

The goal of the game server was to be able to update multiple players, the tile map, and abilities in real time whilst ensuring that the players remain synchronized. This was achieved by ensuring reliable message delivery and by sending full snapshots as a fail-safe. Player movement was implemented on the client side for simplicity, whereas an optimal solution would be server-authoritative to prevent cheating.

Some optimization problems, such as latency compensation, were not implemented but they were researched and could have been implemented on top of the architecture. The main result for networking was two different messaging protocols.

The project was able to cover the most important aspects of the presented networking theory and can be considered a success. One of the goals was to find simple solutions, but each solution came with some drawbacks. For an indie developer trying to compete with larger companies with more manpower, it is recommended to use existing tools that have been optimized for the purpose.

The scope of the game was too large, considering the limited toolset. For research purposes, the choice of tools sufficed for the thesis, but for a commercial game the performance of the program was not optimal. In conclusion, developing networked games takes more time, which means features must be cut if the project should ever be completed. The result was a technical demonstration, but not a complete playing experience.

Lua was not designed for large code bases in mind, which is the reason why statically typed languages such as C# are commonly used for larger projects. Therefore, some of the difficulties such as inheritance were also linked to the language, and to improve performance it could be recommended to use a lower level language that allows memory management.

References

1 Haazen Daniël (2017) How to LÖVE: Learn How to Program Games with the LÖVE Framework. [Online] Available at: https://sheepolution.com/learn/book/contents (Accessed 15 January 2020)

2 Roberto Ierusalimschy, Luiz Henrique de Figueiredo and Waldemar Celes (2006) Lua 5.1 Reference Manual. [Online] Available at: https://www.lua.org/manual/5.1/manual.html (Accessed 12 June 2019)

3 Roberto Ierusalimschy (2003) Programming in Lua (first edition). [Online] Available at: https://www.lua.org/pil/contents.html (Accessed 12 June 2019)

4 Diego Nehab (2004) LuaSocket Documentation. [Online] Available at: http://w3.impa.br/~diego/software/luasocket/home.html (Accessed 11 November 2019)

5 David Murray, Terry Koziniec, Kevin Lee and Michael Dixon (2012) Large MTUs and Internet Performance. [Online] Available at: https://www.researchgate.net/publication/235257878_Large_MTUs_and_internet_performance (Accessed 6 February 2020)

6 Subhash Chandra Yadav and Sanjay Kumar Singh (2009) Introduction to Client Server Computing. New Age International Ltd.

7 Bob Nystrom (2014) Game Programming Patterns. [Online] Available at: https://gameprogrammingpatterns.com/contents.html (Accessed 5 September 2019)

8 Difference Between Client-Server and Peer-to-Peer Network (2017). [Online] Available at: https://techdifferences.com/difference-between-client-server-and-peer-to-peer-network.html (Accessed 5 February 2020)

9 Joshua Glazer and Sanjay Madhav (2016) Multiplayer Game Programming: Architecting Networked Games. Addison-Wesley Professional.

10 Jean-Pierre Corriveau and Jacob Agar (2014) Dead Reckoning Using Play Patterns in a Simple 2D Multiplayer Online Game. [Online] Available at: https://www.hindawi.com/journals/ijcgt/2014/138596/ (Accessed 10 February 2020)

11 Alyosha Pushak (2017) Client-side Prediction for Smooth Multiplayer Gameplay [Online] Available at: https://www.kinematicsoup.com/news/2017/5/30/multiplayerprediction (Accessed 5 November 2019)

12 Valve Developer Community: Source Multiplayer Networking [Online] Available at: https://developer.valvesoftware.com/wiki/Source_Multiplayer_Networking (Accessed 5 November 2019)

13 Glenn Fiedler (2008) UDP vs. TCP [Online] Available at: https://gafferongames.com/post/udp_vs_tcp/ (Accessed 5 November 2019)

14 Gabriel Gambetta: Fast-Paced Multiplayer (Part I): Client-Server Game Architecture [Online] Available at: https://www.gabrielgambetta.com/client-server-game-architecture.html (Accessed 10 February 2020)

15 Peter Andreasen (2018) Deep dive into networking for Unity's FPS Sample game - Unite LA [Online, Youtube] Available at: https://www.youtube.com/watch?v=k6JTaFE7SYI (Accessed 7 September 2019)

16 Abdelkhalek Ahmed, Bilas Angelos and Moshovos Andreas (2003) Behavior and Performance of Interactive MultiPlayer Game Servers [Online] Available at: https://www.researchgate.net/publication/239932670_Behavior_and_Performance_of_Interactive_MultiPlayer_Game_Servers (Accessed 8 March 2020)

17 “No Bugs” Hare (2015) On Cheating, P2P and [non-]Authoritative Servers from “D&D of MMOG” [Online] Available at: http://ithare.com/chapter-iii-on-cheating-p2p-and-non-authoritative-servers-from-dd-of-mmog-book/ (Accessed 8 March 2020)

18 Stevens W. Richard (1994) TCP/IP-Illustrated. Addison-Wesley Professional.

19 Stateful and Stateless Applications Best Practises and Advantages [Online] Available at: https://www.xenonstack.com/insights/stateful-and-stateless-applications/ (Accessed 8 March 2020)

20 Merkes Matt (2014) What is a Callback, Anyway? [Online] Available at: https://www.codefellows.org/blog/what-is-a-callback-anyway/ (Accessed 8 March 2020)

Simple Traversal of UDP over NAT

A typical listen server setup is depicted in Figure 1. Player A is hosting the server on Host A, and Player B on Host B tries to connect.

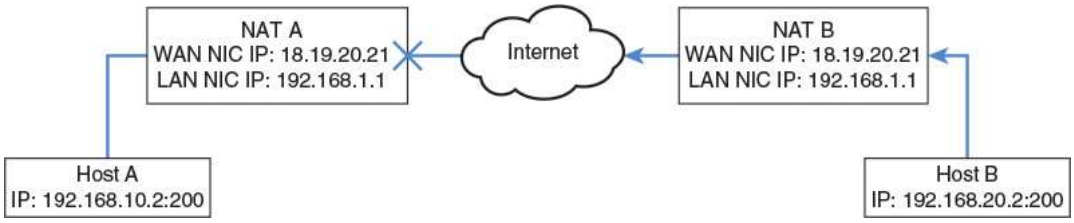

Figure 1. Typical user gaming setup [9, ch. 2]

As illustrated in Figure 1, Host B is unable to connect because there is no NAT table entry for the incoming message. Thus, the router does not know where to forward the packet. The router can be configured so that incoming messages to a specific port are forwarded to Host A. [9, ch. 2]

Simple Traversal of UDP over NAT (STUN) solves the problem without requiring any technical knowledge from the users. It is based on using a third-party server to automatically configure the NAT tables for Hosts A and B to enable direct communication.

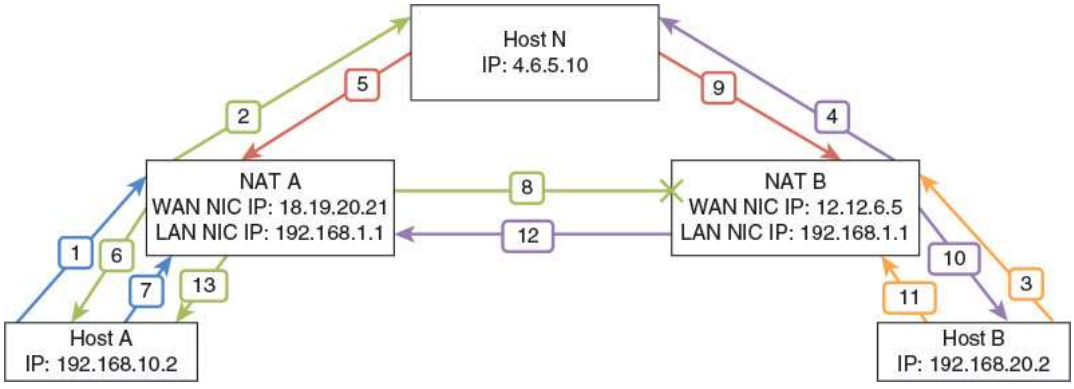

Figure 2. STUN data flow [9, ch. 2]

Host A sends a request to Host N to register as the game server (1), which adds Host N to the NAT table and allows Host A to receive messages from Host N (2).

Host B sends a request to Host N to register as a client (3), which allows Host B to receive messages from Host N (4).

Host N informs Host A of a new client (5), which then learns of Host B's public address and port (6). By sending a packet to the public address (7), Host A adds Host B to the NAT table (8), allowing Host A to receive messages from Host B.

Host N informs Host B (9) of Host A's public address (10), which is now ready to accept packets. Host B is now able to send messages directly to Host A (11, 12, 13).

Shared message handling

```
function Connection:handleMessage(dataString, ip, port)
    if dataString then
        local decoded, data = pcall(Data.decode, dataString)

        if decoded and self:validRequest(data, ip, port) then
            self.updates:handleHeaders(data.headers)

            if data.headers.batch then
                self:handleBatch(data, ip, port)
            else
                self:handlePayload(data, ip, port)
            end
        else
            print("Error: invalid message")
        end
    end
end

function Connection:receive()
    while true do
        local data, msg_or_ip, port = self.receiveFunction(self.udp)

        if data then
            self:handleMessage(data, msg_or_ip, port)
        elseif msg_or_ip ~= 'timeout' then
            error("Network error: "..tostring(msg))
        else
            break
        end
    end
end
```

```
function Connection:update(dt)
    self:receive()

    self.tickrateTimer = self.tickrateTimer + dt

    self:checkTimeout(dt)

    if self.tickrateTimer >= self.tickrate then
        self:sendUpdates(dt)

        self.tickrateTimer = self.tickrateTimer - self.tickrate
    end

    self.state:update(dt)
end
```