UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Centre de la Imatge i la Tecnologia Multimèdia

Development of an Online Videogame using Steamworks API

Jonathan Cacay Llanes

Director: Mónica Martín Mínguez

Degree: Videogame Design and Development

Academic Year: 2024 - 25

University: Centre de la Imatge i la Tecnologia Multimèdia

Acknowledgments

To my friends and classmates,

for always being there and for all the adventures we have lived together.

To my tutor Mónica,

for her patience, guidance and support.

To my mother, father, my sister and her partner,

for their unconditional support.

And to you for reading this.

Thank you.

Index

Summary

To develop an online videogame it needs to take into account smoothness and low-latency experiences equally for all players who are interacting in real-time. This final degree project initially addressed the problem of netcode optimization by designing and implementing a custom networking solution for a cooperative zombie shooter game.

This initial approach allowed an understanding of low-level networking techniques such as snapshot interpolation, delta compression, and custom synchronization of the networked objects. However, halfway through the project, the custom netcode was replaced with Unity's Netcode for GameObjects (NGO) framework to ensure the maintainability and scalability that it offers. While the custom netcode provided valuable insights and control that was taken into account for the project, adapting the project to Netcode for GameObjects offered a more robust base.

The main goals of the project is to develop a functional online prototype for up to 4 players capable of synchronizing entities and player actions while using Steamworks API for its features. The methodology implemented is through iterative development and testing, while using tools like GitHub for version control and Trello for project planification and management.

The final prototype will be a functional videogame with the combination of Unity's Netcode for GameObjects with Steamworks, by showing a responsive and fluid and low-latency multiplayer experience that maintains synchronization.

This project provides a detailed case study for other developers in case they are willing to implement a cooperative 4-player based zombie shooter while contributing to the game development community by documenting the process in this document.

Resumen

Para desarrollar un videojuego online es necesario tener en cuenta tanto la fluidez como experiencias de baja latencia de forma equitativa para todos los jugadores que están interactuando en tiempo real. Este Trabajo de Fin de Grado abordó inicialmente el problema de la optimización del netcode mediante el diseño e implementación de una solución de red personalizada para un juego cooperativo de disparos con zombis.

Durante el desarrollo, a mitad del proyecto, el código de red personalizado se sustituyó por Netcode for GameObjects (NGO) de Unity para garantizar el mantenimiento y la escalabilidad.

El código de red personalizado proporcionó información valiosa y control que se tuvieron en cuenta para el proyecto, pero fue necesario adaptarlo a Netcode for GameObjects, ya que ofrecía una base más sólida.

El objetivo principal del proyecto es desarrollar un prototipo jugable en línea para hasta 4 jugadores en un nivel generado aleatoriamente en un entorno donde se pueda sincronizar todo tipo de entidades, acciones de los jugadores. Todo esto con la ayuda de las funcionalidades que ofrece Steamworks API.

La metodología en la cual se trabajará con el prototipo, estará compuesta por desarrollo por iteraciones, donde se utilizará Github para el control de versiones y Trello para la planificación y gestión del proyecto.

El prototipo final será un prototipo funcional que combina Netcode for GameObjects de Unity con Steamworks, por tal de ofrecer una experiencia multijugador que se note fluida, de baja latencia y con sincronización.

Este proyecto proporciona un estudio de caso detallado para otros desarrolladores en caso de que deseen implementar un shooter cooperativo de zombis para 4 jugadores, contribuyendo a la comunidad del desarrollo de videojuegos mediante la documentación del proceso en este documento.

Keywords

Netcode for GameObjects

Multiplayer

Unity

Steamworks

Links

Initial project (Sockets): https://github.com/Ethanm-0371/Networks_Project

Final project (Netcode for GameObjects): https://github.com/xGauss05/TFG_Project

Video: https://www.youtube.com/watch?v=hSOmpQTEKgA

Tables Index

Figures Index

Glossary

Acknowledgments: Data sent between client and server to confirm the reception of data packets, with the objective to ensure reliable communication.

Actor: Any object that can be placed within the game world or scene in Unreal Engine.

Bandwidth: Quantity of data that can be transmitted through a network in a given time. Commonly measured in Mbps (megabits per second).

Client-Server: A network architecture model where a server manages the communication and synchronization with multiple clients.

Data stream: Flow of data transmitted between the client and the server.

Dead reckoning: Technique used to predict the future position of an object based on the velocity and direction, which reduces lag in online games.

Dead zone: Area in which the movements or inputs of a player are not generating response due to the network settings.

Deterministic lockstep: Technique used to synchronize the world state, based on sending the user inputs.

Entropy coding: A method that compresses data in frequency patterns to reduce the size of a packet.

GameObject: Base class for all objects that can exist in a scene. It is the fundamental object type in Unity. They represent characters, props, scenery, etc.

Geometric compression: Technique used to reduce the size of geometric data to optimize the transmission over the network.

Headless application: Application that runs without a user graphical interface. It works in the background without any visual components.

Hit registration: Process in which the server confirms if a shot has reached its target, which can be affected by the latency and the lag compensation.

Input validation: Process of verifying input data provided by the client to prevent errors or exploits.

Instance-based multiplayer: Architecture in which the players are connected in a unique instance of a server instead of a persistent world.

Interpolation: Technique that calculates values between the received data to smooth transitions and movements in online games.

Interpolation delay: Time period in which the client delays the visualization of the data to smooth out the experience and reduce jump positions due to the latency.

Interpolation period: Time interval during which interpolation occurs.

Jitter: Variation in the arrival times of data packets, which can cause inconsistencies in the gameplay.

Lag compensation: Technique that adjusts the execution time of actions as a function of the latency that the player has so that the events occurring affect all the players equally.

Latency: Time that takes a packet to travel from the client to the server and vice-versa.

MMORPG: Massively Multiplayer Online Role Playing Game. A game genre where a large number of players interact in a persistent virtual world.

Netcode: Code responsible for managing the network communication in a videogame, which includes synchronization and packet managing.

NetworkObject: A GameObject in Unity which enables it to respond and interact to the netcode.

Object replication: Process of copying and synchronizing objects between client and server to maintain consistency of the game's state.

Object serialization: Process to transform an object into a format that can be sent over the network and then reconstructed back on the receiving end.

Packet duplication: Error in the data transmission in which the data packets are sent and are being processed more than once, causing undesired effects.

Packet header: Section of data at the beginning of a network packet that contains important control information regarding the packet.

Packet loss: Loss of data packets during transmission. Can cause desynchronization or errors in the gameplay.

Packet ordering: Technique used to make sure that the packets sent are being processed in the correct order while minimizing synchronization inconsistencies.

Packet period: Time interval between the sending of consecutive data packets in a network connection.

Peer-to-peer: Network model in which clients send data directly with each other without an intermediary server.

Ping: Diagnostic tool to measure latency and connectivity. In multiplayer games, measure the time between the sent packet and the received packet. Expressed in milliseconds.

Prediction error correction: system used in online games where the server corrects any differences between the client prediction and the server reality.

Prefab: A template of a GameObject in Unity that allows reusability in different parts of the game.

Reliability: Network property that ensures the delivery of all the packets sent.

RPC: Remote Procedure Call. Method that allows executing functions on a remote system. Frequently used to synchronize events between client and server.

RTT: Round Trip Time. Total time it takes for a packet to travel from the client to the server and vice versa.

Rollback netcode: Technique used in networking, especially in fighting games, to reduce input lag and improve responsiveness of online play.

Scalability: Capability of a program to handle any increase of users or data without losing performance.

ServerRpc: A remote procedure call (RPC) that can be only invoked by a client and will always be received and executed on the server/host.

Socket: Endpoint of a network connection. Used to send and receive data between devices.

State synchronization: Process of aligning the state of the objects, players, and physics from both server and client sides.

Static zone: Area in a game that does not change its state and/or behavior during gameplay.

TCP: Transmission Control Protocol. A network communication protocol that ensures reliability over speed.

Thread: A program that allows it to be performed in parallel. Useful for network processes for online games.

Tickrate: Frequency in which the server updates the game state and sends it to the clients. Commonly measured in ticks per second.

Timestamp: Time record that marks the exact moment a data packet is sent or received.

UDP: User Datagram Protocol. A network communication protocol that prioritizes speed over reliability.

Unity: A cross-platform game engine developed by Unity Technologies. Used to create 2D and 3D games.

World state: Representation of the whole state of the game which includes objects, players, and physics.

Wrapper: Interface that is used to transfer specific data through the network.

1. Introduction

Online videogames development has been evolving significantly in the last years, but one of the challenges that developers face is network optimization. In an online environment which some videogames offer, where many players interact in real-time, problems such as latency, network performance, synchronization and many other different aspects are crucial for every single client connected since it can negatively affect the user experience. As more complex the game is, the better solution needs to be developed to have a stabilized connection.

This document aims to research different techniques and technologies that a developer can apply or implement in a project that requires implementing netcode. It focuses especially on network optimization. By implementing algorithms alongside advanced techniques of optimization, the project will focus on the communication between server and clients, by reducing the amount of latency, optimizing the information sent through the network and applying efficient methods to synchronize the data sent or received. All of this will be taken into consideration when developing the project with the pretense to offer a likeable user experience that does not affect the gameplay.

The project will focus on the creation of a client-server model that allows scalability and performance in dynamic environments. It will be limited to a 4 player experience so that the amount of data handled is the minimum possible so that we can focus on the data sent by the players.

To have a prototype, the project will be worked alongside Ethan Martín which will be researching a different topic but will be taken into consideration in the development of the project: procedural level generation.

This work not only will have a technical insight, but will also feature balancing between the gameplay and the efficiency of the netcode, which is crucial to online videogames to obtain a fluid experience.

1.1 Motivation

Since I was 8 years old, I have been playing many videogames, which are mostly MMORPGs (Massively Multiplayer Online Role-Playing Games). These games presented an immersive world where thousands of players were connected across the world. I explored a wide variety of MMO games, which made me more curious on how a multiplayer videogame is made.

For this reason, my curiosity towards the development and technology that created these types of videogames, was growing which made me want to create my very own online videogame. How do the servers that handle many players work? How is the information transmitted efficiently and synchronized across all the players? These two questions have been on my mind since I grew curious towards the online games and became the foundation of the project at hand.

As of today, one of my dreams is to contribute to the creation of an MMORPG as a game developer or as a network developer. I want to develop a multiplayer game to start down this road.

1.2 Problem Statement

In online multiplayer games, the smoothness and the reliability for the player's interactions, dynamic objects and interactive environments like in MMORPGs or cooperative shooters, we need to ensure that the netcode's performance is as much optimized as possible since it is the system in charge of synchronizing the actions, maintain consistency and minimize the latency.

There are many online videogames out there that have a netcode implementation, but still the stagnation of having bad connection, desynchronization between action and input, and other topics regarding latency are still there.

There are many existing solutions for online videogame development. This includes pre-built libraries or frameworks (for example, Photon Engine, Mirror Networking). These tools provide a working solution, however lack the flexibility to obtain a more customized case for a specific project. For example, Valheim, a survival and sandbox videogame by Iron Gate Studio, initially used a peer-to-peer model with Unity's UNET but later it was switched to a client-server model due to the scalability issues. Another example would be Overwatch 2, a shooter game produced by Blizzard Entertainment where they developed its own rollback netcode instead of relying on pre-built solutions, since the existing options, frameworks or tools could not meet the precision, requirements and responsiveness required for a competitive first person shooter.

If that is the case, why are there still games that are online focused have issues regarding latency which causes discomfort among the audience? There aren't many open source

solutions which makes it difficult for developers to know what is the best approach in order to start developing a networked game and end up with a combination of various techniques and algorithms.

With games that include procedural content generation, AI-driven behaviours, and real-time updates, we need a more customized solution. We can work on the robustness and the latency in more detail, but we must consider that a poorly implemented netcode might result in issues such as desynchronization of players, actions delayed or random disconnections, an issue that some of the popular games are still having.

For this project, where the focus is on creating a cooperative zombie shooter with procedural mapping, addressing the netcode differently while manually managing it through the code is the main topic that this project will focus on.

This is the reason that this project is for; controlling the game's data exchange efficiency and aim for a low-latency performance. And as a personal reason, to gain a more knowledgeable basis to create an online game.

1.3 General Objectives of the Project

The final goal of this Project is to develop and implement an online cooperative multiplayer game which features procedural generation.

This project aims to:

- Create a functional prototype of a cooperative online game that features a procedural environment and real-time interaction for up to 4 players.
- Handle dynamic objects, synchronize player actions and maintain consistent game states.
- Create an experience where a player can invite up to 3 more players to enjoy the game.
- Adapt a cyclic level generation into a multiplayer game.
- Efficiently apply Steamworks API to ensure that everything works correctly.

1.4 Specific Objectives of the Project

To achieve the general objectives, these specific objectives need to be achieved firstly:

- Synchronize the data regarding the procedurally generated map so that all the connected clients can interact with the same information.
- Offer a smooth experience and efficient synchronization regarding the net objects (dynamic objects), entities and player interactions.

- Document the whole development process to provide a guideline on how I made it work.

1.5 Scope of the Project

The project will be focused on the development of the functionalities that a multiplayer game offers. As for the development of the procedural map generation, it will not be part of this scope but only the transmission of the generated content.

The project will be a functional prototype of a 4-player cooperative game that features:

- Synchronization between player actions, AI-based zombies, and net objects replication.
- Procedural map data transmission and synchronization only.

The objective is to demonstrate a working multiplayer experience rather than delivering a completed game with animations, cinematics, etc.

This project is directed to anyone that might show interest towards it which are:

- Players / Gamers: those who like playing cooperative multiplayer games that offer real-time interaction.
- Developers: for those who are exploring or investigating custom networking solutions for online videogames.

The project will not only will benefit the developers who have developed the project, but also will contribute to videogame sector where professionals or particulars can use this document to apply it to their own projects. By offering a practical case about how to design and implement a network cooperative videogames, by focusing on the functionalities that a multiplayer game offers.

- Indie developers or big studios: this project can serve as a reference for developers that are looking for a solution that can be applied to their projects, such as the techniques used on certain implementations. Mostly focused on cooperative shooter games.
- Research and development teams: it might be useful for those teams that are in charge of R&D who are investigating new multiplayer videogames.
- Single developers: those who are working on their own and are looking for a personalized solution can find this document and project to use it as a base for their own multiplayer experience.

Regarding the players, who are also benefiting from the advancement that this document gathers, might indirectly affect the gameplay experience that they encounter when playing

online cooperative videogames. Regarding the techniques utilized, players can enjoy the following:

- Gameplay experience: players who enjoy cooperative gameplay might encounter multiplayer cooperative videogames that have applied the techniques that this document gathers such as the reduction of latency or the synchronization that players might experience. Even with an efficient IA driven entity, that in this case it is zombies, could be synchronized in a precise way, so that players can not experience desynchronization issues.
- Real time stabilization: shooter games that feature cooperation, are benefitted specially on the precise synchronization, where players cooperate with each other and coordination is crucial with the player actions. A low latency and optimized netcode is key to obtain real time synchronization and stabilization.

2. Foundations and Context of the Project

2.1. State of the Art

The development of online multiplayer videogames has improved over the past years, by focusing on providing low-latency solutions for the players. This has been influenced by innovations in the network architecture, algorithms for latency compensation, and real-time data synchronization techniques.

This section will explore the current state of the art in network optimization for online games. Technologies, methodologies, and industry practices that are being used, by providing an understanding of the advancements and challenges that the project might encounter.

2.1.1. Specific techniques and solutions for Online Videogames

State Synchronization

- Deterministic Lockstep

  A synchronization technique that to synchronize the world state only the input done by the players will be sent and will be replicated in all instances of the game to replicate the same action.

  The main advantage of applying this technique is that the packet sent does not contain a lot of information, which means that the size of said packet will not be big and it does not depend on how complex the game is, since only that input the game will be synchronized.

  Although, every technique has its disadvantages, and this one is no exception; The game needs to be deterministic, meaning that the inputs and world states will need to give an entry and always will give the same outcome. For example, in a Chess game, when you move a piece, both the player and the opponent will know where that piece will end up. So, the problem happens when physics is applied; the outcomes will be different since the time step of every client might vary from machine to machine.

  An example of a game that uses deterministic lockstep, would be *StarCraft,* a real-time strategy videogame where the player controls military science fiction characters. In this game, hundreds of troops are being simulated, and synchronizing all the units will be hard, but since it is a deterministic videogame, by only knowing the input, synchronizing all of them is more than enough to synchronize the world state.

- Snapshot Interpolation

This technique consists in capturing the moment in a state all the player states and everything related to it, then sending it as a packet so that every player will receive it and interpolate the last state received to the new one. Unlike deterministic lockstep, this can not be used for deterministic videogames, but the size of the packet grows if many states the player changes or needs to be updated, which includes changes in position, velocity, etc.

The main advantage that snapshot interpolation offers, is that the synchronization is somehow perfect. So, for example, losing a packet that contains information, does not matter that much since the next time it receives a packet, the state will be interpolated with the new information received.

But this technique also comes with a disadvantage. The size of the packet increments as much new information is added, which can generate latency because it congests the network. There are some techniques that can alleviate this congestion or to reduce the size of the packet sent through the network.

- State Synchronization

  A technique that combines both deterministic lockstep and snapshot interpolation; we send both the input and the state in a single packet.

  Now that we have seen both advantages and disadvantages from the previous techniques, the state synchronization technique we send both inputs and states through the network. This allows us to send updates for some objects, and if we apply some other techniques, we can select which objects will be sent per packet, allowing us to save bandwidth by adding in a single packet the updates on the most important objects.

  State synchronization uses the best feature that the previous techniques offer, which can deliver to the players a smooth and solid gameplay experience.

State Prediction

- Rollback Netcode

  This technique is used mostly on fighting games, such as *TEKKEN 8* because the order of actions done is important. This technique consists in predicting what is the next action that the opponent will execute so that the gameplay feels smooth and exact. The moment that the prediction fails, we rollback a few frames back before the prediction so that later the action will be executed correctly.

- Client-Side Prediction

  Although this technique is like the rollback netcode, it is used in a more generic way. All clients connected will predict which next state will be from the other clients. In case the prediction is not the correct one, it will execute a state interpolation to the state received from the server.

Packet Compression and Optimization Techniques

- Snapshot Compression

  Snapshot compression is a series of techniques that are key to optimize the transmission of world state between server and clients. The objective is to reduce the size of the packet sent to achieve a better efficiency of the bandwidth and reducing the latency. Applying this technique should not affect the precision and the real-time synchronization. But, to apply this compression, some techniques will need to be applied to achieve this.

- Delta Compression

  Applying delta compression, is just sending the deltas towards the previous snapshot. This allows us to reduce the size of the packet since we are only sending a single value. Instead of sending the position of a player that is standing still, no move at all, no additional data will be sent regarding its position.

- Quantization

  From the values that some data can have, we round the float numbers to a more discrete number. It will have low precision but reduces drastically the size of the data sent. Instead of sending a position value of 32 bits, we can send a discrete value of 16 bits, which is 2 times less than the earlier.

- Entropy Coding

  They are a series of compression algorithms that assign smaller codes to data that have a higher probability of appearing and larger codes to data that have a lower probability of appearing. The objective, as the other mentioned techniques, is to reduce the quantity of bits sent needed to represent the data.

One of the techniques to apply in videogames, is the Huffman Encoding, that applies the entropy coding principles in a Huffman Tree. Basically what it does is ordering the symbols by frequency in a priority queue in which it combines the 2 symbols with lower frequency in a node, assigning them “0” and “1” branches. It repeats this process until the whole tree is completed.

- Dead Reckoning

  It is the process of predicting an object’s behaviour based on the assumption that it will keep doing whatever it is currently doing. For example, if a player is walking forward, this process will assume that the player will keep going forward. However, when the players take different actions than the one assumed by the prediction, the client will diverge to the real state and will correct it.

  This technique tries to guess the right state most of the time, but if it guesses wrong, it should readjust to obtain the desired behaviour.

- Spatial Partitioning

  Spatial partitioning is a useful technique since it divides the game in different zones and only sends the relevant data for each client depending on their position. This allows to minimize the data sent to the client by reducing the unnecessary information. For example, a player that it's being separated in an isolated area from the other players can only receive information regarding the objects in its surroundings, without affecting the other players. This technique is mostly used in the MMORPG genre which handles hundreds of players in a server instance.

2.1.2. Study case in Videogames - Final Fantasy XIV

Final Fantasy XIV (commonly abbreviated to FFXIV) is an MMORPG developed and published by Square Enix. It was originally launched in 2010, but initially was being received negatively, it was relaunched and revived in 2013 under the title of Final Fantasy XIV: A Realm Reborn.

At first, the design was widely criticized for its design problems and performance in 2010, but under a new direction in 2013 by Naoki Yoshida, also known as Yoshi-P, revived the initial title that nowadays FFXIV has become one of the most played MMORPG worldwide. Since then, the game has received multiple expansions, such as Heavensward, Stormblood, Shadowbringers, Endwalker and the latest one as of 2025, Dawntrail, which amplifies the world, narrative and in-game mechanics.

Its in-game mechanics include instances, challenging raids and a system oriented in Player vs Enemy (PvE), but as every single multiplayer game has its issues, Final Fantasy XIV is no exception. At launch, the game has a built-in 300 milliseconds latency, meaning that it made it very difficult for players to react to in-game events. This made that some in-game mechanics in raid content where quick reactions were very crucial to avoid were almost impossible to avoid on reaction. This was something that needed to be fixed as soon as possible, so in a later update, the development team reduced the innate latency to 100 ms, which aligned closely to the industry standards. While this became a vast improvement over 300 ms, players were still complaining about not being able to avoid or dodge some attacks.

The feeling of getting hit by attacks that should have missed them was largely due to the communication between the game server and the player's client. Nevertheless, FFXIV's player base often assumes that when an attack animation starts, that is when the attack will hit. But it is determined when the cast bar of an ability is over when the ability will hit, not the visual animation that some attacks create. The game processes the cast bar finishing, and if the player targeted is within the attack's radius when that happens, the game would deem that as being hit, even if the animation has not yet completed. The server calculates whether a player was hit based on their position at the moment an ability finishes its casting bar.

Figure 1: Final Fantasy XIV - Telegraphs

While these animations are kind of the appeal that FFXIV offers, in the context of an MMO, it becomes a problem, especially for attacks that are real quick or have delayed effects. The acclaimed title could alleviate some confusion by reducing the attack animations or making them more instantaneous to improve communication to players and show them when attacks

are actually happening. This is commonly referred to as phantom hits, where players believe they have dodged the attack but still take damage.

This implies client-side and server-side discrepancies. A player's position is registered in the user's client, but the server has other information. This creates an illusion when two players see each other in different positions due to the latency, leading to a situation where one player appears to be doing the in-game mechanic correctly while the other client sees the same player in another spot, or even worse, behind on the server information. Some of the aspects that the game offers, like tethers, debuffs or spread mechanics are not done correctly for the fact of not registering correctly even though the players are positioned correctly but causing an undesired behavior.

Final Fantasy XIV, like any other MMORPGs, are in a server live-state or client-server architecture, where most of the game actions need to be validated before reflecting on the client side.

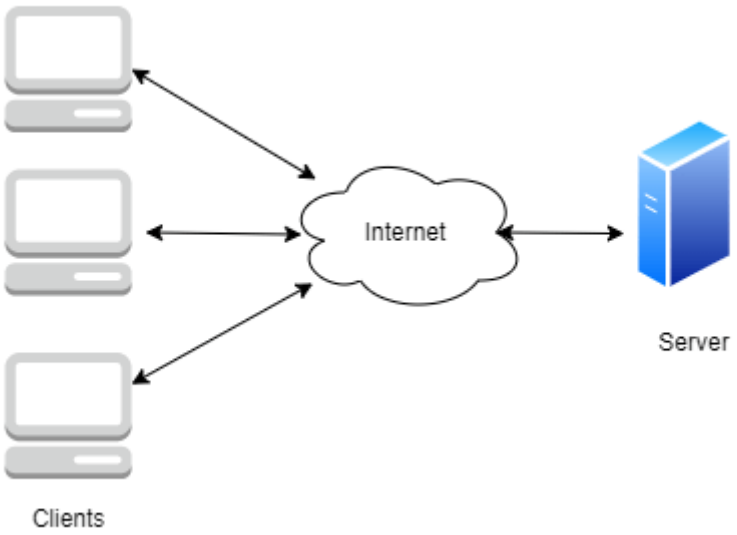


Figure 2: Client - Server diagram

When a user gives the order of 'Jump', the client would send the command to the server. Then, the server processes the request, acknowledges it and then sends back the response to the client. When the client receives the acknowledgement, the jump animation would render on the client's screen.

If there is latency and the user tries to attack a monster in the game, the attack command would be sent to the server, but to the latency, the action will not be executed instantly. Instead, the monster who is controlled by the server in real-time, would keep attacking the user, regardless of the latency of the player, thus registering all the damage dealt. When the

connection establishes, the player's attack would be processed, causing a delay on the execution of actions.

In case of an AoE of a monster, the server would send the information with an exact timestamp. Due to the client being a little bit behind the server, the AoE is live on the server. If the user is suffering from latency and has not received the information regarding the attack. As soon as the information received, the monster would appear casting the ability and the AoE telegraph would also appear. The difference is, that the user would be desynced with the server, meaning that if the client managed to step outside of the telegraph, in the server the player position might be outdated and the damage would be registered. If the client does not send the command to move to the server to update the player position before the AoE ends, the player would be hit by it. This is a common issue on most multiplayer games that need to be lightened to offer a fluid experience.

In order to send the packets of information to control the game from server to client, and client to server, FFXIV uses 3 types of packets:

- Base Packets

Base packets are the main packet container for all the subpackets. Basically, it contains a number of subpackets to be processed. They begin with a byte header, which means that the header of the packet consists of 16 bytes of data. The structure of the header is as follows:

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Is authentic ated	Is compressed /encoded	Connection type		Packet size		Number of subpackets		Timestamp in milliseconds							

Table 1: Final Fantasy XIV Base packet structure

"**Is authenticated**" is the byte information for the initial handshake. It is either 0 or 1. 0 when the packet is the initial handshake, and 1 when not.

"**Is compressed/encoded**" depends if the client is communicating to the lobby server or the map server. If it is the lobby server, it indicates that the packet is currently encrypted using Blowfish (a codified protocol of symmetric blocks designed by Bruce Schneier in 1993), or if it is a map server, the packet is compressed using zip compression. Server also can set this byte to 0 and send directly unencrypted or uncompressed packets and have them read properly.

"**Connection type**" will be flagged depending on the connection type it is trying to establish. In FFXIV, there are 3 types: Zone Connection (0x1), Chat Connection (0x2) and Lobby Connection (0x3). Once the server responds, this will be set to 0x0.

"**Packet size**" is the total size of the packet including the header.

"**Timestamp in milliseconds**" as its name implies, is a unix timestamp written in milliseconds.

- Segment header

Segment packets are smaller packets that are in the base packet. Each one of them begins with a 0x10 byte header. The structure of the header is the following:

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Segment size		Type		Source ID				Target ID				Unknown			

Table 2: Final Fantasy XIV Segment structure

"**Segment size**" is the total size of the subpacket including header.

"**Type**" is the field that defines what this subpacket does. 0x0 is the initial handshake and 0x3 is a game packet, but the other codes are practically unknown and not found out what they do. An assumption would be that some of the types are zone server related, but not confirmed.

"**Source ID**" is the number identity of the actor that triggered this packet.

"**Target ID**" is the number identity of the actor that this packet is for.

"**Unknown**" are bytes that are reserved for something that FFXIV uses it for, but it is unknown.

- Game Packets

From the subpackets, if the type is 0x3, that means that the packet is a game packet. These are the main game packets that both the lobby and map server use to control the game logic. And like the other packets, they do also have their own headers of 0x10 bytes in size. But, the only difference is that these packets are harder to decipher. The only thing that is confirmed is that 4 bytes are used for the timestamp, and 2 bytes for the opcode, which defines what the game packet does.

All of this information can be checked using a packet analyzer, such as Wireshark, which is an open-source tool used for network troubleshooting, analysis, software and communications protocol development.

Figure 3: Final Fantasy XIV - Telegraphs 2

But, alongside these undesired behaviors, the FFXIV developer team has made real good implementations regarding the server partitioning that it features. Servers are grouped by data centers, or the physical servers where they are hosting the information. These servers include North America, Europe, Oceania and Japan. A user has to choose a server depending on how closer the user is towards the physical server to experience less network lag. And each of these servers are partitioned into different data centers in order to avoid congestion and high levels of traffic.

2.2. Theoretical Framework

In this section, a foundation will be provided to understand some of the concepts that implies developing an online game with a focus on network optimization.

2.2.1. UDP or TCP?

Before deciding on which protocol to use, defining both concepts is a good start to decide which protocol is going to be used depending on the implementation of the project. Both protocols have a different way of sending the packets.

TCP

TCP (*Transmission Control Protocol*) establishes a connection between 2 devices before the data transfer occurs. It guarantees the data transfer without occurring any errors, since it uses a confirmation system and packet retransmission of packets lost. This is a good option for

applications that require high reliability, but at the same time it requires a higher latency since it requires establishing a connection and confirming it.

UDP

In contrast with the previous protocol, UDP (*User Datagram Protocol*) does not establish a connection with the other end before transferring data, and the data is sent through datagrams through the network. A datagram is an independent data packet that is sent through the network.

This protocol is faster than TCP since it does not require the confirmation of the packet transferred and no packet retransmission for those packets lost. This also means that some data can be lost or arrive in a different order in which they are sent, which can cause problems for those applications that require reliability.

In summary, UDP is a faster option but less reliable in comparison to TCP. In online videogames, a combination of both is frequently used. UDP for the transmission in real time like the position of the players, important objects, etc. and TCP is used for example starting a session or sending a message through the in-game chat.

Protocol	TCP	UDP
Connection	Connection-oriented	Connectionless
Usage	High reliability, critical-less transmission time	Fast, efficient transmission, small queries, huge numbers of clients
Data packets order	Rearranges packets in order	No order
Data stream	Read as a byte stream	Sent and read individually
Reliability	Yes	No
Error checking	Error checking and recovery	Simply error checking, no error recovery
Acknowledgement	Acknowledgement segments	No acknowledgement

Table 3: TCP vs UDP comparison

The correct answer in which network protocol to use, would be a mix of both. Use UDP for those packets that need to be sent as soon as possible, such as player actions, enemy actions or anything that takes into account the world state of the videogame. Use TCP for those

packets that do not require to be sent quickly, but at the same time does not lose during the transmission. These actions would include spending in-game currency, paying for in-game currency, sending chat messages, or even a whole implementation of a turn-based game where speed does not matter.

2.2.2. Network Architecture

Network architectures are also important to decide to develop the netcode. They are important because they manage how the different clients will transfer the data between each other to maintain world state synchronization and to achieve a smooth gameplay. Some of the different architectures have their own advantages and disadvantages. The most used ones in online videogame development are the following:

Client-Server

The client-server architecture is used in videogames where the players connect to a centralized server that manages the game, which means that it oversees the synchronization of the world state and sends the information to all the connected clients. All game clients send their inputs to the server. This is also known as authoritative servers and dumb clients. Everything that happens in the world is being monitored by the central server.

According to Gabriel Gambetta, in his article "*Client-Server Game Architecture*", it all starts when the client who acts as a server starts cheating. As a cheating player, not only is making the experience better for himself, but they are also making the experience even worse for all the clients who are connected to the server. As a videogame developer, you have to consider avoiding all kinds of cheating methods since it tends to drive the players away from the product you are trying to deliver.

Gabriel Gambetta defines the authoritative servers and dumb clients, in which the server controls everything that is happening in the game while the clients are able to modify the environment. With this way, if any client attempts to cheat, for example, his damage output, the server knows exactly the damage that the client does, which can correct it directly. The game state is managed by the server alone, while clients send their actions to the server. Then, the server updates the game state periodically and sends back the new game state towards all the clients.

Peer-to-peer

In the Peer-to-peer (commonly referred as *P2P*), players connect directly between each other without the need of a centralized server. Every client is responsible for sending their inputs and to synchronize their world state to the other clients. This architecture has its flaws, since it can cause security problems and might be more difficult to manage if there are a lot of players.

But there is also the option to combine the client-server architecture with the peer-to-peer which is the hybrid P2P connection. In this combination, a client assumes the role of being the centralized server where he is authoritative towards the other clients connected. As developers, apply different techniques to switch the host that has the most stable connection from the others so that the experience is more doable as equal among the connected clients.

Client-Server	Peer-to-peer
Clients and Server are differentiated	Clients and Server are not differentiated
Focus on information sharing	Focus on connectivity
Centralized Server stores data	Each peer has its data
Server responds with the Clients requests	Each and every connected peer can request and respond
More stability	Less stability if increased connected peers

Table 4: Client-Server vs Peer-to-Peer comparison

2.2.3. Network Common Problems

Some of the concepts that network developers might encounter during the development of the netcode of the videogame will be defined next.

Latency

It is the time elapsed in between an initial state and the next state. It is also referred to as the time elapsed in between packets. Latency is unavoidable, and depending on the genres there are certain thresholds that latency should not surpass. For example, VR games are the most sensitive to latency because we humans expect to see the things, we see the moment we move or tilt our head. According to the book, *Multiplayer game programming: Architecting networked games* by Glazer Joshua & Madhav Sanjay (2015), latency can range from 16 to 150

ms before the user starts to feel that the game is sluggish and unresponsive, but depending on which type of game the user is playing, latency can grow as high as 500 ms without being detrimental to the user's experience.

To calculate the time difference in between packets, we calculate the RTT (round-trip-time). It is the time that it takes to send a packet to the server and return.

But latency can also occur even if not related to network issues, such as the input delay that the user has, or even the render of certain models or objects on screen from client-side.

As a network developer, our aim is to decrease the amount of latency so that the users' play experience improves.

It needs to be considered that there are many sources of latency, which can be the packet transmission that is usually the most significant cause of latency. These are the delays that affect the packet transmission latency:

- Processing delay: as its name suggests, it is the delay that a network router device reads to know which device needs to send the packet and then forward it.
- Transmission delay: it is the time spent in order to write the information to the physical medium, which is the cable.
- Queueing delay: a single router device can only manage a finite amount of packets at a time. If many packets are being processed at the moment, and a new packet arrives at the router, a waiting time is being processed.
- Propagation delay: is the time that it takes a packet to reach the destination through the physical medium.

Although some of these delays can be optimized by the network developer, some of them cannot be managed. For example, the propagation delay depends on the length of the wires and the distance between connected clients that are currently exchanging information. The only thing that the developer can do, in this case, is to move the hosts close together, or even use region-based routing so that a player can only find players from its own region.

Jitter

Refers to a deviation of the RTT from an expected value. This deviation is caused by several occurrences that might cause network delays. A high deviation might provoke that the packets will arrive in an order that they are not supposed to be, causing failures that might affect the synchronization between players and world state. For example, players warping back and forth due to packets arriving in a different order.

The aforementioned delays in the latency section can contribute to jitter. The processing delay can occur during the router processing the packets. Transmission and propagation can also contribute to jitter since these two delays depend on the route that the packet passes through. And the queue delay might also vary since depending on what is happening in the game, a big amount of packets can be sent towards the clients which can overwhelm the router and cause a queue of packets.

Packet loss

As its name implies, it is when the packet sent is lost and does not reach its destiny. This is caused by both internal and external factors. Like network congestion, bad connection quality or even a malfunctioned ethernet cable.

Some of the causes that packet loss might occur is when the physical medium is malfunctioning. As developers this is something that we cannot avoid and needs to be handled by the user. Also, link layers have also predefined rules in whether which data can be sent or cannot be sent. And when the router has a crowded queue, depending on which router the client has, it might skip some packets causing packet loss.

Losing an important packet might cause a bad experience between players. For example, if a position packet is lost, it does not matter that much since it can be fixed by using state interpolation, but when it comes to shooting it affects the gameplay experience, since it can decide which player wins.

Desynchronization

Desynchronization occurs when the received data by the players does not match between them. This can be caused by a bad netcode or a combination of the earlier mentioned problem concepts. A desynchronization causes the game to be negatively affected and makes it harder for the players to interact between each other.

It is an important problem that needs to be solved as soon as possible since the different players will start seeing different realities because of the data mismatch sent from the clients.

2.3. Contextualization

The foundations of multiplayer videogames and their netcode was already established and standardized. The multiplayer experience has been improving through the addition of new techniques, code improvements, physical assets and many more advancements that helped improve the gameplay experience in multiplayer videogames.

Many of the techniques used in videogames are the ones that are mentioned in the previous section and become standardized in many published titles. To this point, where multiplayer videogames are pretty much known worldwide, a lot of effort has been put into further improving the experience with the new techniques, protocols, money invested, according to Shockbyte James Zinn, in 2022 alone ten billion dollars in tech-related spending from gaming companies was solely for multiplayer services.

As for the most notable multiplayer videogames as of 2025, some titles that are worth mentioning are Counter-Strike 2, Fortnite, Minecraft, ROBLOX, Marvel Rivals, League of Legends, Dota 2 and VALORANT. These games are, as of 2025, the current leaders in the online genre and have gained their popularity due to the fact that they are online experiences, and obviously, their innovation in gameplay and their player interaction.

Online gaming has been in a constant evolution, improving netcode management, client synchronization, latency reduction and even the implementation of newer techniques and technologies that companies have researched on their own. This allowed developers to offer even smoother experiences with more precision in multiplayer games, which gave the popularity of the videogames mentioned above.

2.4. Market Study

The online videogames industry has experienced an increase in recent years, by becoming one of the most dynamic and lucrative in the entertainment industry. With the advancements in network solutions and the expansion in PC, consoles and smartphones, online videogames have achieved millions of players worldwide.

The Spanish online market has stayed firm in its position year by year as one of the key contributors of the future of the videogame industry. In 2023, its revenue saw a growth of around 120 million euros. One of the reasons for this increase could be the rise in popularity of online platforms, which have once again become the main source of income. These platforms have generated around 715 million euros in 2023 compared to 581 from 2022. Subscription services to access the multiplayer features such as PlayStation Network or Xbox Live, contributed in this situation, reaching nearly 80 million euros in 2023.

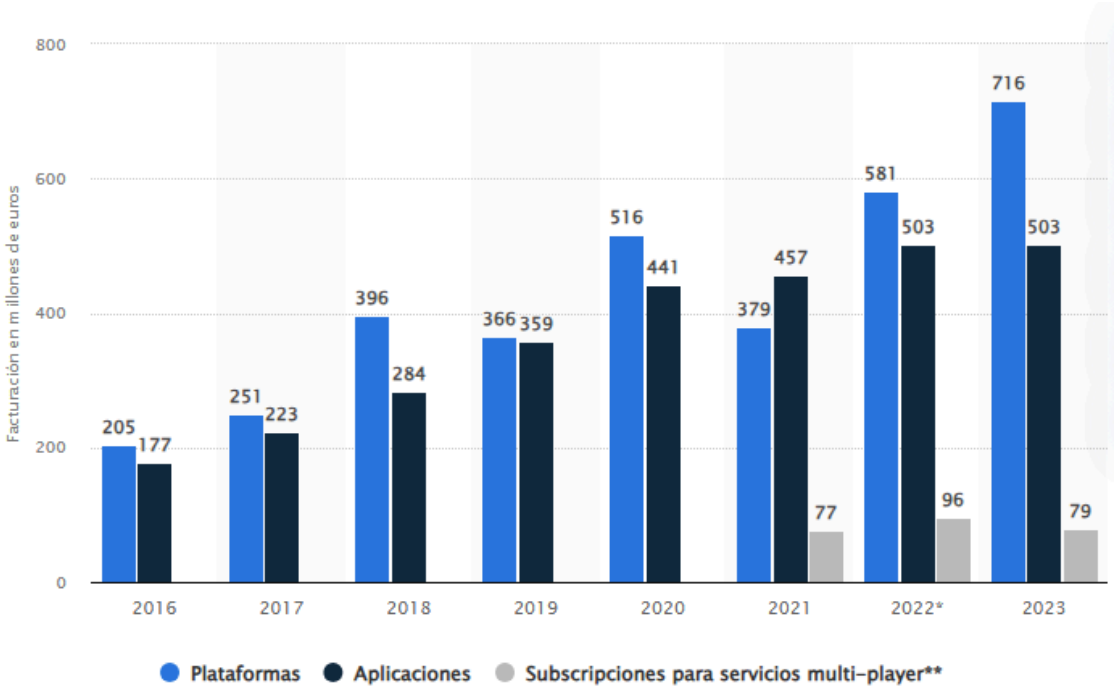


Figure 4: Spanish turnover in millions, over the years in the Videogame Industry

In 2023, the global online gaming market generated approximately 26.14 billion U.S. dollars in revenues, which translates to a 9.8% growth compared to the previous year. In 2020, online gaming revenues increased by a significant amount due to the COVID-19 outbreak. This forced many people to stay at home and had no other choice to consume digital content as entertainment and find other sources or ways to connect with others via online services.

As of 2025, online gaming is still a popular activity and currently, there are an estimated 1.1 billion online gamers worldwide where most of them are located in China, South Korea, and Japan where they have the biggest online community the rest of the population.

Online gaming has already been evolving over the past years. From classic PC and console-based online gaming genres such as massively multiplayer games (MMO gaming) and competitive first-person shooter.

As mobile connectivity and smartphone ownership increased in recent years, so did online mobile gaming which allowed newer audiences who do not have to rely on PC, consoles, or any subscription to play with other people. Cross-platform titles with focus on mobile are currently the ones pushing the online genre forward. The most popular online gaming format right now is battle royale games such as PlayerUnknown's Battlegrounds (PUBG), Fortnite, and Apex Legends generated tens of millions of players within months of their release

2.4.1. Game Engines

Unity

One of the popular game engine platforms for developing videogames offers a networking system that allows developers to create their own multiplayer videogames in an efficient and flexible way.

Unity's networking system features both architecture models, peer-to-peer and client-server. For starters and beginners, the game engine features 3 example netcode projects that anyone can download and tamper with however they want, from a 3D casual co-op game with production level code, to a 2D co-op space adventure that uses Netcode for GameObjects to help the users synchronize network objects.

Working in a client-server topology in Unity it is required to spawn NetObjects (objects that require to be updated in the server), which is different from spawning objects in a single-player instance. By Unity's network API to spawn an object, we need to make sure that it is made as a Prefab, which is a combination of components, data, scripts, etc. that the NetObject will be using.

While Unity is written in C++, developers who are tampering with it can not access the source code of the engine. JavaScript was used in the past in order to write the code Unity, but nowadays C# is the language most used in the engine.

As an addition, Unity has a package that helps managing entities in the videogame, called Netcode for Entities. And to monitor network performances, Unity allows installing multiplayer tools such as the Profile, Runtime Stats Monitor, Network Simulator, and Network Scene Visualization to optimize the videogame that the developer is trying to implement, which allows seeing the game's performance and usability. All of this if the developer chose to use Unity transport. If any other transport is used, the developer must find other ways to monitor their traffic.

Also, has support for remote procedure calls. In Unity, a command is an action sent from a client to the server, while a client RPC function is an action sent from the server to a client.

Moreover, Unity's networking library provides matchmaking functionality which can be used to request and list the current game sessions. Once a session that is suitable for the client is found, then the client can join the game.

Games like Pokémon GO, Among Us, Escape from Tarkov, Genshin Impact, Hearthstone are made using Unity.

Unreal Engine

Unreal Engine offers a network system designed to help developers easily in the development of an online videogame. According to the developers of Unreal Engine, their architecture is based in replication and client-server.

As a difference with Unity, Unreal Engine also creates their own specific terms, such as Actor, which can be the equivalent of GameObject in Unity.

In the game engine, actor replication is the name for the process of synchronizing data and procedure calls between clients and servers. This system provides a higher-level abstraction alongside low-level customization to allow developers to deal easily with all the situations that a developer can encounter while creating an online videogame.

It even offers a complete documentation on how this system works and online tutorials that anyone can access for free so that they can implement their own online videogame.

Unreal Engine has a system of remote procedure calls, that in Unreal are: server, client and multicast.

- Server function: called on a client and executed on the server.
- Client function: the contrary of the server function; server calls a client function, then the procedure call is sent to the client.
- Multicast function: a function that is called on the server and executed on both clients and server.

Games like Fortnite, Rocket League, VALORANT, ARK: Survival Evolved, PUBG: Battlegrounds are made using Unreal Engine.

GameMaker

GameMaker offers online functionalities, but as a difference from Unity and Unreal Engine, its networking system is even more manual. The network communication is mainly done using sockets and messages, which gives more control to the developers to control what is being sent or received, but requires even more work from part of the developer. At the same time, users can select either TCP or UDP when doing their implementation.

The data packets sent are made up of binary data taken from a buffer. This buffer can be created and manipulated by using the buffer functions that GameMaker offers.

Also, GameMaker has built-in functions such as *network_send_udp* or *network_destroy* to help developers in the process of connecting sockets.

Like the other game engines, it offers documentation so that developers can create their online videogames while reading the documentation.

Games like Nidhogg, Realm of the Mad God, Rivals of Aether are made using GameMaker.

Godot

Godot offers a networking system integrated that allows developers to create their own videogames online or even LAN games. It uses the client-server architecture with tools to manage the data replication and synchronization between players.

Has support over standard low-level networking via UDP, TCP and some higher-level protocols such as HTTP and SSL. These protocols are flexible and can be used for almost anything.

Since Godot scripting can be done by either GDScript, a programming language that is used in the engine, or in C#, Godot offers built-in functions to help developers create an online videogame.

Nevertheless, they also have their own documentation that anyone can access for free to help them in the process, where every built-in function is explained.

Although Godot does not have many prominent AAA videogames, games like Thrive, BitBrawl, Pixel Wheels are made using Godot.

2.4.2. Videogames and their Network Implementations

Left 4 Dead (2008) & Left 4 Dead 2 (2009)

One example of the most representative in efficient netcode is Left 4 Dead & Left 4 Dead 2, developed by Valve. This multiplayer cooperative videogame up to 4 persons, was created using the game engine Source, which is the game engine that Valve created in 2004 that only features the client-server networking architecture, that limited the possibilities in which Source could help develop multiplayer videogames.

Multiplayer games based on Source, a client connects to a game server, which communicates with the server and vice versa by sending small data packets in a high frequency. Then, the client would receive the world state from the server and generate video and audio output depending on the updates received. Clients also send data from input devices back to the server to process the data received and send the according updates. Also, clients can only communicate with the game server and not between clients that are also connected to the server (not a peer-to-peer architecture).

In terms of network bandwidth, Source was very limited in this aspect; the server can not send new update packets to every client for every single world state change. According to the Source Multiplayer Networking wiki from the Valve Developer Community, the server takes snapshots of the current world state at a constant rate and broadcasts it to the clients, which take a

certain amount of time to travel between the client and the server. This means that the client time is always a bit behind the server time. In addition, the client input packets are also delayed on their way back, so the server is processing temporally delayed user commands.

So, to fix all these problems or at least alleviate any delay, Source applied techniques such as data compression and lag compensation. Then, the client would predict and interpolate the received updates to achieve a fluid experience.

Left 4 Dead & Left 4 Dead 2, offered a 30 tickrate, which means that the timestep was about 0.033… seconds. This means that every update from the server occurs every 33.3 milliseconds, which is crucial for the entity interpolation, input prediction and the lag compensation that the Source game engine offers.

So, the crucial optimizations and how are implemented, are as follows:

Entity Interpolation

Clients receive about 20 snapshots per second, that means that every 0.05 seconds, or 50 milliseconds, a client receives a complete update from the game state, while, as mentioned earlier, the server processes the game logic in ticks of 33.3 milliseconds. In this case, clients would interpolate between snapshots to mitigate and achieve a fluid experience to avoid subtle movements.

Source engine default interpolation period is of 100 milliseconds. Even if a snapshot is lost during the packet processing, there will always be two valid snapshots to interpolate between.

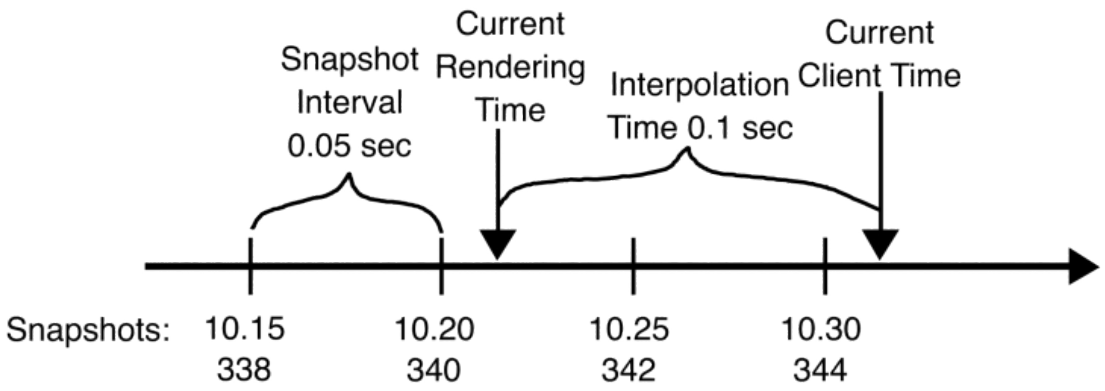


Figure 5: Entity Interpolation - Arrival times of incoming world snapshots

In Figure 5, the last snapshot received was at tick 344 or 10.30 seconds. Then, the client time continues to increase based on this snapshot and the client frame rate. The rendering time would be then, if the current client time is 10.32, minus the view interpolation delay of 0.1

seconds (delay default given by Source), the client would see the animations interpolated in the 10.22 second.

Input prediction

Source engine takes into consideration the network latency that clients can have. If a client has a network latency of 150 milliseconds and tries to send an input key, the user command then would be processed and the client's character would recreate the action that the input key has given. This process would be delayed by the 150 milliseconds that the client has, which means that this delay will be applied to all the client's actions.

Prediction in this case in the Source engine, is only possible for the view of the local client, since prediction only occurs on the client side. In this case, the client's key presses and will try to "guess" where the player will end up. Predicting other player entities would mean predicting the future without any information, so there is no possible way to do this.

Lag compensation

In Source, their lag compensation implementation keeps history of all the recent player positions for about 1 second. A newer user command would estimate at what time the command was created in this way:

Command Execution Time = Current Server Time - Packet Latency - Client View

This only applies to only the players. The user command is executed and the action sent would be detected correctly. After a user command is processed, players would then revert to their original positions.

Although lag compensation is a hard technique, there is always room for a small error that can hinder the experience. As a developer, this inconsistency problem can not be solved in general due to the fact that packets can have different speeds.

World War Z (2019)

World War Z features an online cooperative experience up to 4 players that uses a hybrid network implementation where it offers dedicated servers and combines peer-to-peer communication to offer a multiplayer fluid experience.

Their netcode implementation is not public, so every single technique that it is being discussed is purely by intuition and experience from the player themselves that have played this title.

The only information regarding their implementation, is that it is created by around 100 people of Saber Interactive inspired by Left 4 Dead, by using a proprietary game engine named Swarm Engine, which helped render the huge zombie hordes that it features.

Although every single implementation of netcode has its flaws, Saber Interactive has an issues article that any user can read if they are experiencing basic connection issues, which includes a small guide on how a player should proceed against these problems.

Remnant: From the Ashes (2019)

Remnant: From the Ashes, a game created by Gunfire Games. netcode has been a topic of debate in the player base. Although there is not so much public information regarding their implementation of the network, many players have discussed the architecture model they used, state synchronization issues and some problems related to latency in the cooperative mode.

But something that is confirmed, is that they use a client-server model since it requires you to connect to a dedicated server in order to play with anyone. It does not connect directly with another client, which means that it is not a peer-to-peer model.

According to Steam general discussions forum and Reddit posts, there have been problems with their network connection implementation, resulting in random disconnections from the clients. This suggests that potential problems and inefficiencies are happening in the game's netcode, and we can possibly assume that it is due to an unoptimized client-server connection implementation.

Even talking about input delay, when playing cross play with a client who connects through PlayStation 4 and a PC client, the PlayStation 4 user would then feel input delay on their actions which created an unbearable experience.

Many users have stated that it is better to play this game in a single-player experience than playing cooperatively with someone else due to the fact that it is, according to a Reddit user, painful, especially in higher difficulties.

Back 4 Blood (2021)

Back 4 Blood, a videogame created by Turtle Rock Studios in which some of the developers that helped develop Left 4 Dead, worked on this title. It featured a cooperative zombie experience up to 4 players that gained notoriety thanks to having worked previously with Left 4 Dead. Due to this reason, users had high expectations, but Back 4 Blood failed to achieve the same level of success that its predecessor achieved.

Some users claim that one of the problems that Back 4 Blood has is the lack of an impactful story campaign, something that Left 4 Dead managed implicitly in the gameplay.

Back 4 Blood was implemented in Unreal Engine 4, a game engine developed by Epic Games released in March 2014. Unreal Engine 4's source-code is available for a monthly subscription of US$19, including all the C++ files and scripts. Like their predecessors, it uses a client-server model where the client who hosts, is run locally in their machine.

At launch, the game did not feature an offline feature, meaning that players could not play the videogame without being connected to their servers, but was fixed after 3 months of its release.

2.5. Differentiation

Nowadays, online videogames do have different strategies for the optimization of the network connection. As seen previously, the most common ones are the interpolation, the packet compression and the efficiency in managing events and state synchronization. Big companies have made their own engine and even implemented their network functionality, but are not that public for anyone to see the implementation. This project aims to offer a source-code available to the public so that anyone who is interested can read. This is the difference between a published game from a company and this project.

3. Project Management

In this section, the tools that the project will use are mentioned below in order to have an organized project and to keep track of the progress.

Since it is a Unity project, we need an online repository, in order to keep track of the changes made into the project, see what changes have been made in order to track all the possible errors that might occur during the development process. For my case, the tool that is going to be used is GitHub, a version control software that allows hosting Git repositories online.

Then, to keep track of the project milestones, Trello will be used, a tool that basically manages projects through tables, cards, by having three different columns: to-do, doing and done. Cards will be distributed depending on the state of the feature with its colored tag and in the different iterations.

For programming stuff, we need to design it before getting hands-on work. For this purpose, draw.io will be used as a tool that allows me to create diagrams of any type. With this tool, it will be used to create flux diagrams, interaction diagrams and class diagrams. If a quick sketch is needed to gather and organize the ideas, Microsoft Paint will be used to visualize it.

Now that the tools have been covered to keep track of the progress of the project, following the next topic, to validate the project it is needed to test the prototype. Since the game engine that is being used is Unity, in their documentation there is a section regarding testing multiplayer games locally.

To test it, a playtest is needed in order to stress test multiple instances of the videogame so that we can test a multiplayer session between players. In addition, in the coding part, developers need to add debug tools in order to keep track of packets that aren't sent (packet loss), the amount of ping that the user has in milliseconds. To test it even further, alongside my partner we will be testing the connectivity between me and him to test if the gameplay feels fluid. In case that my partner is not available, in Unity we can simulate up to four players simultaneously on the same development device by installing the Multiplayer Play Mode package, or use third-party tools such as ParrelSync, which is a Unity editor extension that allows users to test multiplayer gameplay without the need of building the project.

3.1. SWOT

This section evaluates the strengths and weaknesses, and the opportunities and threats that a project regarding a network-optimized online game project can encounter in its development.

	Positive	Negative
Internal Origin	**Strengths** - Technical focus on Network Optimization - Covers both theoretical and practical aspects regarding network optimization. - Collaboration with a teammate that will be specialized in procedural generation.	**Weaknesses** - Complexity in implementing and optimizing network algorithms. - Potential difficulties in balancing academic deadlines with the project. - Learning advanced network optimization techniques in the given timeframe.
External Origin	**Opportunities** - Contributing to open-source networking solutions and publishing the research found. - Use of this project as a stepping stone for job opportunities or internships in the Networking area of the videogames.	**Threats** - Technical challenges in network synchronization that might delay the overall progress. - Evolving technology that might make this solution obsolete. - Limited resources and testing environments that can hinder the validation of the project.

Table 5: SWOT Table

3.2. Risks and Contingencies plan

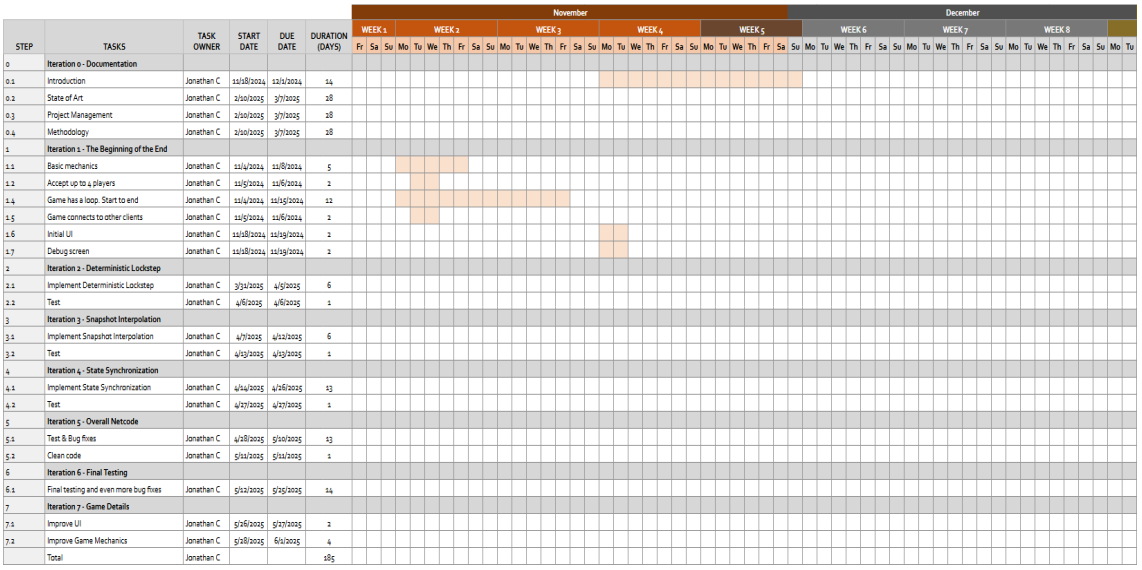
In any project, problems might arise when least expected; this project is not exempt from it. This is why having a contingency plan for any given problem is essential to mitigate possible issues that might delay the progress of the project. This section covers some of the risks and the corresponding strategies and/or solutions that might prevent, manage, and overcome the problems without harming the project itself.

Risk	Solution
Lack of proper documentation regarding the features implemented in the project that might delay certain milestones.	Dedicate a specific time to document the features implemented for each milestone.
Balancing the amount of work in the academic environment and deadlines that might delay the project's progress.	Prioritize core functionalities and/or reduce the scope if necessary while maintaining a functional prototype.
Technology obsolescence, where networking technologies might make some solutions outdated.	Focus on adaptable and well-documented technologies, or even change the current implementation to Photon Unity Networking or any other framework.
Complexity of optimizing the network solution, as if it may be more challenging than expected.	Spend even more time for testing and debugging to refine the performance desired.
Not knowing how to implement a specific feature.	Ask for help from different programmers to glimpse their perspective.
Coordination with the partner. Synchronizing different development aspects from both networking and procedural generation.	Use version control systems, such as GitHub and schedule a periodic meeting to identify, test and resolve conflicts early.
Limited access to network infrastructure for testing.	Search for a better solution online that provides a server service to access a temporary resource.

Table 6: Risk and contingencies

3.3. Initial cost Analysis

3.3.1. Gantt Chart & Hours count



STEP	TASKS	TASK OWNER	START DATE	DUE DATE	DURATION (DAYS)
0	Iteration 0 - Documentation				
0.1	Introduction	Jonathan C	11/18/2024	12/1/2024	14
0.2	State of Art	Jonathan C	2/10/2025	3/7/2025	28
0.3	Project Management	Jonathan C	2/10/2025	3/7/2025	28
0.4	Methodology	Jonathan C	2/10/2025	3/7/2025	28
1	Iteration 1 - The Beginning of the End				
1.1	Basic mechanics	Jonathan C	11/4/2024	11/8/2024	5
1.2	Accept up to 4 players	Jonathan C	11/5/2024	11/6/2024	2
1.4	Game has a loop. Start to end	Jonathan C	11/4/2024	11/15/2024	12
1.5	Game connects to other clients	Jonathan C	11/5/2024	11/6/2024	2
1.6	Initial UI	Jonathan C	11/18/2024	11/19/2024	2
1.7	Debug screen	Jonathan C	11/18/2024	11/19/2024	2
2	Iteration 2 - Deterministic Lockstep				
2.1	Implement Deterministic Lockstep	Jonathan C	3/31/2025	4/5/2025	6
2.2	Test	Jonathan C	4/6/2025	4/6/2025	1
3	Iteration 3 - Snapshot Interpolation				
3.1	Implement Snapshot Interpolation	Jonathan C	4/7/2025	4/12/2025	6
3.2	Test	Jonathan C	4/13/2025	4/13/2025	1
4	Iteration 4 - State Synchronization				
4.1	Implement State Synchronization	Jonathan C	4/14/2025	4/26/2025	13
4.2	Test	Jonathan C	4/27/2025	4/27/2025	1
5	Iteration 5 - Overall Netcode				
5.1	Test & Bug fixes	Jonathan C	4/28/2025	5/10/2025	13
5.2	Clean code	Jonathan C	5/11/2025	5/11/2025	1
6	Iteration 6 - Final Testing				
6.1	Final testing and even more bug fixes	Jonathan C	5/12/2025	5/25/2025	14
7	Iteration 7 - Game Details				
7.1	Improve UI	Jonathan C	5/26/2025	5/27/2025	2
7.2	Improve Game Mechanics	Jonathan C	5/28/2025	6/1/2025	4
	Total	Jonathan C			185

Figure 6: Gantt chart - hours planification

Each of the squares are a minimum of 4 hours of work, and the range of the project lasts for about 185 days in total. Weekends are also taken into account since we are doing half day. Since there is no specific job position to be working in the network systems of a videogame, for the calculation of the salaries we will take into account the salary of a junior software developer. According to Indeed, he amount of hours that a junior developer in networking systems will get paid in Spain as of 20th of March, 2025, ranges from 11.50€ ~ 20.25€ per hour, where 185 days will be used to develop the project, and these days at least 4 hours will be spent to work by applying the following formula:

Import = Days * Hours * Price per hour

The import amount sums between 8.510€ ~ 14.985€ gross.

A detailed breakdown of development milestones can be seen in the Annex section of the Gantt chart. There, the amount of hours can be seen in Figure 6, where every single square marked equals 4 hours of work and specific the estimated completion hours in which the tasks will be completed.

3.3.2. Network Infrastructure

To host a server dedicated to the project, Amazon Web Services will be used (or AWS), Cloud Computing. And to connect to this server, PuTTY will be used, a free SSH and telnet client for Windows.

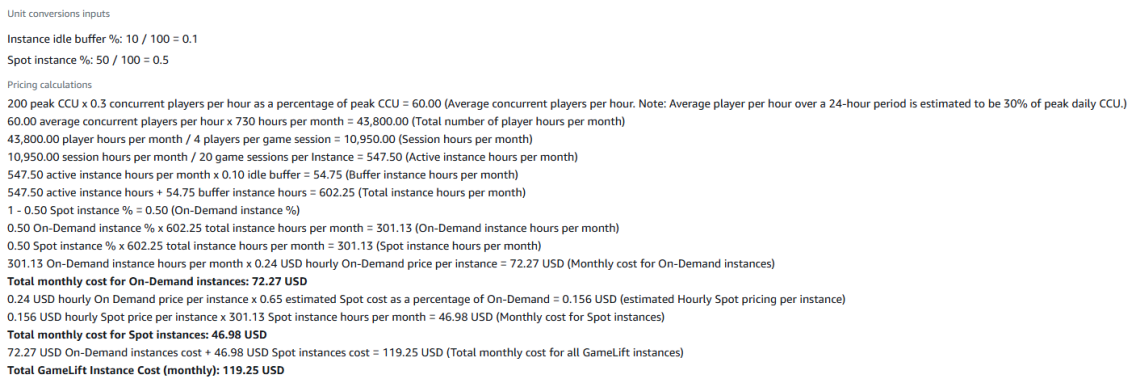
For the scope of this project, AWS offers a 12 month free service with limited functionality, but the amount of features that it offers is more than enough for what I need for this project.

But when the project finally releases, the game server should be moved to Amazon GameLift, which is a service from Amazon Web Services that gives many additional features specialized for game hosting and matchmaking. By using the price calculator that Amazon offers, by using the following attributes:

- Region: Paris
- Peak concurrent players (peak CCU): 200
- Game sessions per instance: 20
- Players per game session: 4
- Instance idle buffer %: 10
- Spot instance %: 50
- Instance type: c5.large (vCPU: 2, Memory (GiB): 4)
- Operating system: Windows

The total costs reach to:

- Total monthly cost for On-Demand instances: 72,27 USD (66,75 €)
- Total monthly cost for Spot instances: 46,98 USD (43,39 €)
- Total GameLift Instance cost (monthly): 119,25 USD (110,15 €)


Unit conversions inputs

Instance idle buffer %: 10 / 100 = 0.1

Spot instance %: 50 / 100 = 0.5

Pricing calculations

200 peak CCU x 0.3 concurrent players per hour as a percentage of peak CCU = 60.00 (Average concurrent players per hour. Note: Average player per hour over a 24-hour period is estimated to be 30% of peak daily CCU.)
60.00 average concurrent players per hour x 730 hours per month = 43,800.00 (Total number of player hours per month)
43,800.00 player hours per month / 4 players per game session = 10,950.00 (Session hours per month)
10,950.00 session hours per month / 20 game sessions per Instance = 547.50 (Active instance hours per month)
547.50 active instance hours per month x 0.10 idle buffer = 54.75 (Buffer instance hours per month)
547.50 active instance hours + 54.75 buffer instance hours = 602.25 (Total instance hours per month)
1 - 0.50 Spot instance % = 0.50 (On-Demand instance %)
0.50 On-Demand instance % x 602.25 total instance hours per month = 301.13 (On-Demand instance hours per month)
0.50 Spot instance % x 602.25 total instance hours per month = 301.13 (Spot instance hours per month)
301.13 On-Demand instance hours per month x 0.24 USD hourly On-Demand price per instance = 72.27 USD (Monthly cost for On-Demand instances)
Total monthly cost for On-Demand instances: 72.27 USD
0.24 USD hourly On Demand price per instance x 0.65 estimated Spot cost as a percentage of On-Demand = 0.156 USD (estimated Hourly Spot pricing per instance)
0.156 USD hourly Spot price per instance x 301.13 Spot instance hours per month = 46.98 USD (Monthly cost for Spot instances)
Total monthly cost for Spot instances: 46.98 USD
72.27 USD On-Demand instances cost + 46.98 USD Spot instances cost = 119.25 USD (Total monthly cost for all GameLift instances)
Total GameLift Instance Cost (monthly): 119.25 USD

Figure 7: Detailed information from the Amazon GameLift service.

3.3.3. Additional resources

In a project, consider the essential resources to carry out the project such as hardware, software, server infrastructure or any development tools that are required to progress the

project. Although some might be obvious, it is important to know which assets are needed to consider and apply for the project.

To develop this project, a Game Engine is indispensable to develop the project. For this purpose, Unity is going to be used, which requires the use of Unity's Pro license which costs 2.200€ for the annual license. It is required since it offers advanced networking features, such as cloud diagnostics, performance reporting, which are essential for developers in order to test out and optimize the netcode. The free version of Unity lacks these features and other network tools that are needed to develop this project.

Then, we need additional development tools, such as an IDE and a version control software, which both of them are also available for free.

To gather external resources and information that has a paywall in order to access it, such as online courses, books regarding advanced networking or even consulting a professional in the field, it may be important to consider a small fee to dedicate in this aspect.

In Udemy, which is an online and learning marketplace where professionals post their online courses, the price fee ranges from Free ~ 120€ approximately.

Books regarding the networking field can be acquired in the price range from 10.99€ to 49.99€.

And to use as hardware, by using the personal computer which the components are the following:

- CPU: AMD Ryzen 7 3700X - TDP (Thermal Design Power): 35 W ~ 65 W
- GPU: NVIDIA GeForce RTX 3070 - TDP: 40 W (light use) ~ 220 W (under full load)
- Motherboard: Rog Strix B550-F Gaming: 15 W ~ 30 W
- RAM: 2 X 8GB DDR4 Kingston HyperX Fury Black: 10 W ~ 15 W (both modules)
- Cooling: NZXT Kraken X63: 10 W (idle or low load) ~ 15 W
- Storage (SSD + HDD): 6 W ~ 8 W
- Peripherals: 15W ~ 24W (keyboard, mouse, headset)

Which amounts to a total when the minimum consumption is 126W approximately and when the maximum consumption is 382W approximately.

The daily consumption of this personal computer is, when used minimum:

126 W * 4 hours = 504 Wh (0,504 kWh) / day

382 W * 4 hours = 1.528 Wh (1,528 kWh) / day

And per month, assuming that a month is 30 days and the average price (which varies per hour in Spain) where computer will be located from 18:00 to 22:00 is 0.18 € / kWh, would be:

0,504 kWh * 30 days * 0.18 € = 2,72 € / month

1,528 kWh * 30 days * 0.18 € = 8,25 € / month

To summarize the whole block of cost analysis, the table here shows the amount of cost that a project of this caliber might need.

Asset	Minimum	Maximum
Salary		
2 Junior Developers salary in Spain (11.50€ ~ 20.25€ / h)	17.020,00 €	29.970,00 €
Licenses		
Unity Pro license (1 year)	2.200,00 €	2.200,00 €
Network Infrastructure		
Amazon Web Service (AWS)	0,00 €	0,00 €
Amazon GameLift (price / month)	110,15 €	110,15 €
Steam publish	100,00 €	100,00 €
Resources		
Computers	4.100,00 €	4.100,00 €
Energy consumption (price / month)	2,72 €	8,25 €
Udemy courses	0,00 €	120,00 €
Networking books	10,99 €	49,99 €
Total	23.430,99 €	36.539,99 €
Total + Amazon GameLift + Energy consumption	23.430,99 € + 112,87 € /month	36.539,99 € + 118,40 € /month

Table 7: Minimum and maximum costs

3.4. Environmental Impact and Social Responsibility

This section will focus on how this project could affect the environment and society. This will cover the aspects on how impactful are the use of servers, their energy consumption and the social responsibility of the project.

Environmental Impact

A project of this caliber relies on a server, which consumes energy and contributes with the carbon emissions and most of the servers are located in data centers.

Servers consume vast amounts of energy and electricity to power everything in the infrastructure that supports it. They require significant amounts of energy to work.

Many of these structures have thousands of servers and/or IT devices, which all of them require electricity to run and process data-intensive workloads. According to the U.S. Department of Energy, they consume up to 50 times as much energy per floor space of a typical commercial office building. And according to the International Energy Agency (IEA), they account for 1 ~ 1.5% of global energy consumption.

Data centers and data transmission networks are responsible for nearly 1% of energy-related greenhouse gas (GHG) emissions, which contributes to global warming and climate change. According to the Net Zero Emissions by 2050 scenario, these emissions must be cut in half by 2030.

But, most of the emissions in the videogames sector are not generated by the companies but the players; according to Sam Barratt, Chief of the Youth, Education and Advocacy Unit in UNEP's Ecosystems Division and Co-Chair of the UN Higher Education Sustainability Initiative (HESI), the 80% or 90% of the emissions are generated by indirectly by the industry on how it is played and the products.

For example in 2021, Ubisoft has calculated that only 10% of its carbon footprint comes from its own company, with the majority, the 40% of consumed energy by the players.

A key point is the size of the videogames. Downloading a single game, such as Final Fantasy XIV, that as of 2025 it requires 140 GB available space in a PC, consumes a lot of energy. The energy consumption to download a videogame of this size can vary depending on the efficiency of the servers, the network infrastructure and the device where it's going to be downloaded. The energy consumption by GB transferred, according to a study from the Nacional Lawrence Berkeley Laboratory, the data transfer through the Internet consumes between 5 kWh and 7 kWh per terabyte (TB) in wired networks. This value can be bigger in wireless networks. So, if

we do the conversion from 1 TB to GB, we get 1000 GB, and the consumption per GB is around 5 kWh and 7 kWh we can calculate the approximation of the consumption:

$$140 \text{ GB} \times 0.0005 \text{ kWh/GB} = 0.7 \text{ kWh}$$

$$140 \text{ GB} \times 0.0007 \text{ kWh/GB} = 0.98 \text{ kWh}$$

Then, in order to get the amount of CO_2 it generates, according to the International Agency of Energy (IEA), the average global emission of CO_2 per kWh is 0.4. Then:

$$0.7 \text{ kWh} \times 0.4 \text{ kg } CO_2/\text{kWh} = 0.28 \text{ kg of } CO_2$$

$$0.98 \text{ kWh} \times 0.4 \text{ kg } CO_2/\text{kWh} = 0.39 \text{ kg of } CO_2$$

Although, one person does not generate that much, it is significant in the digital consumption context.

For this project, in order to reduce the amount of emissions, as a developer it is needed to optimize the netcode, the network traffic so that it can be reduced which means that indirectly the energy consumption will be reduced. Applying efficient coding practices can also help reduce the processing power requirements which can also minimize the use of energy.

Then, since at least 4 hours will be spent to develop the prototype, it will be done in a computer which will be consuming energy in a variety of tasks, including the processing of data, the maintenance of the Internet connection, and the images rendering on screen. The consumption might vary, but generally it should be around 200 W and 800W when active, or even more, depending on how demanding the videogame that is being developed.

Social Responsibility

Since this project aims to implement a network solution for players or users to enjoy them, we will be promoting in an ethical way that the development of this project will respect the user's privacy and their data protection within the scope that the project handles.

Moreover, in the game environment that it is going to be created, by applying all the objectives, the design of the videogame will try to be fair and balanced in order to create a multiplayer experience that welcomes a variety of audiences. To achieve this, while doing the prototype tests, a survey will be conducted to consider the player’s feedback so that in future iterations, it might be possible to fix undesired behaviors that the users have found.

As for the project, the code will be public for anyone who is interested in learning about network optimization and they want to check the project that this research paper is focused on, which includes all the guidelines to help other developers learn from and build upon the

project. Users who use this project as an inspiration or re-use the netcode, should add in the list of contributions the users who made this project possible.

4. Methodology

4.1. Project Planification

To progress in the development of this project, an interactive, iterative and incremental development approach will be considered. From the beginning, an initial planification will include all the necessary tasks to conclude the project. The project will follow a Scrum-based Agile methodology, with weekly/biweekly sprints depending on the feature to be developed.

By using this methodology, the planification of this project will be divided into the different implementation phases that it is trying to achieve. This provides the maximum flexibility and adaptability while progressing through this project.

The tasks at hand will be divided into different iterations, where the objective is to finish each iteration with a feature implemented and ready to test. This implies a better product than the earlier iterations, which signifies that every iteration means a functional project.

To use this methodology, tracking the project is mandatory, so for that purpose, Trello will be used to keep track of the progress. To create, organize and keep a continuous evaluation of the tasks and iterations. Panels inside the tool are not needed since the board is only for a single developer who is going to work on the optimization part of the network, but requires cooperation and communication with the teammates in order to divide each iteration with his own milestones.

This board will have its cards color-coded, where green means that the task has been done, yellow means that it is in process and currently work in progress, and red means that the task has not been completed and not yet started. This helps to visualize properly the amount of work that has already been completed. Check the Figure 8 below for visual information.

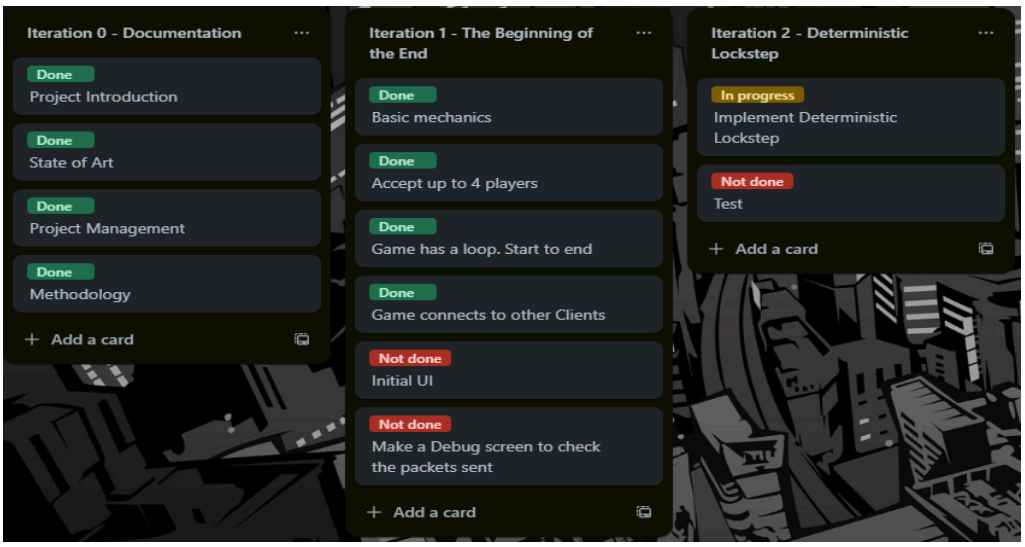


Figure 8: Trello - Initial project iterations planification

For example, in the first iteration of the project, which is the documentation of the theoretical part, cards named “Project Introduction”, “State of Art”, “Project Management” and “Methodology” are part of the Iteration 0 - Documentation.

To control even better the amount of time for each iteration, the Gantt chart can be found in the cost analysis section, or in the Annex with even more detail. The dates that for every iteration will be used to implement a certain feature can be found there. For every single day, a minimum of 4 hours will be spent in order to implement the feature that is being worked during that period. And if we combine the Trello board and the Gantt chart Excel, we can have the different tasks that the different iterations will have and the approximately amount of time it will require to complete each milestone.

During the practical part, the gantt chart or the Trello board might be modified in order to add even more features or functionalities that the project might require, which means to modify the structure and the ranges of the established dates.

Regarding the previous statement, several changes were made to the planification due to a change of objectives for the project. The project itself was focused on the optimization of the network connection, but due to the difficulties of implementing a custom network code, a decision was made to focus more on creating a multiplayer experience using Steamworks API rather than focusing on the optimization of it.

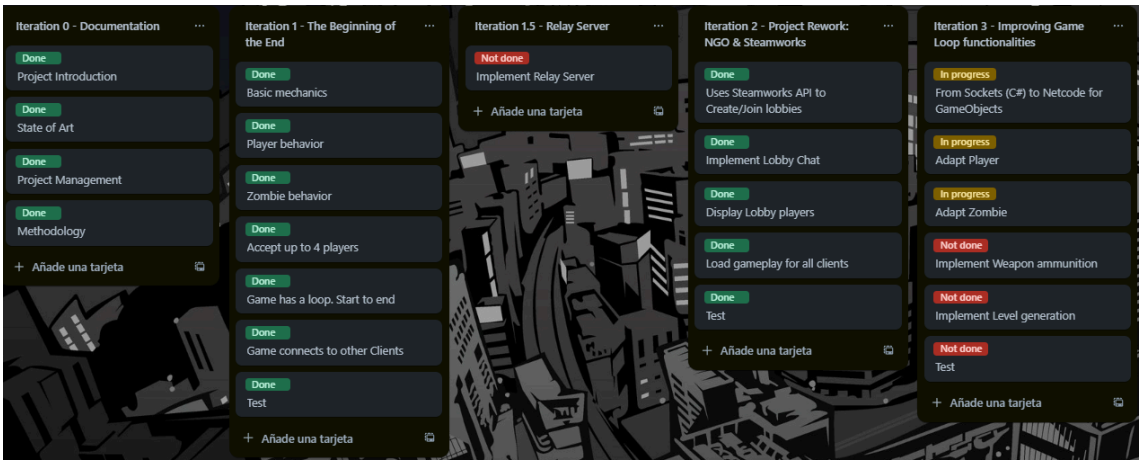


Figure 9: Trello - Current planification

The advancements made from the previous iterations and topic will be used on the new focus, the only thing that needs to be changed is to use Netcode for GameObjects in Unity and combine it with Steamworks API.

Some of the reasons on why the change has been made are the following:

- Complexity of synchronization and game state consistency: maintaining the game state between many clients and a server requires efficient algorithms of interpolation, synchronization and prediction. Without a proper network middleware, all of this must be done from zero.
- Latency and Packet loss management: although latency is something that multiplayer games can not avoid, a custom netcode should include solutions such as lag compensation, which needs to be implemented.
- Scalability and security: creating a secured system requires validations from server-side, protection against network attacks and potentially infrastructure relay, such as relay server.

With the newly adopted approach, Amazon Web Services will no longer be used as a relay server, and the associated costs have been removed from the project budget. Instead, the project will now rely on the Steamworks API, which is free during the development phase. However, when the game is published on the Steam store a fee of 100€ must be paid.

Some of the advantages of using Steamworks API combined with Netcode for GameObjects, which are going to be considered, are the following:

- Efficient integration using Steam platform: Steamworks API allows the developer to create a safe authentication and matchmaking, where lobbies can be established so that players can enjoy a multiplayer experience. Also it allows the integration of Steam friends invitations and achievements.
- Network management using Netcode for GameObjects: by using a modular and extensible package prepared to handle network connections, synchronizing objects, RPCs (remote procedure calls) that are being used for communication between clients and server, authority of objects and reduces the complexity of managing the network communication so that the developer can focus on the game logic.

Aspect	Steamworks API	PlayFab	Amazon GameLift
Solution	Publisher and social API, multiplayer network	Backend as a service for videogames	Servers and sessions management
Objective	Steam integration: achievements, matchmaking, friends, P2P	Users managements, economy, cloud save, multiplayer	Host and automatic scalability for online videogames
Scalability	Not managed	Managed through Azure	Advanced scalability and load balance
Cost	Free. 100€ to publish on Steam	Freemium. Cost per use.	Cost per instance, traffic, storage, etc.
Multiplayer integration	Steam Networking (P2P or client-host)	No networking included.	No networking included.
Authentication	SteamID and Steam accounts	Multiple providers, such as Steam, Xbox, PlayStation…	Requires implementation

Table 8: Comparison between Steamworks API, PlayFab and Amazon GameLift

From the table above, although *PlayFab* is excellent from multiplatform games with a complete backend and *Amazon GameLift* is good for high demanding games that require dedicated servers, the best option for this project is Steamworks API, since not only is completely free in production, but also provides a free ecosystem inside the platform, which provides the Steam integration.

4.2. Theoretical Part Validation

To validate the theoretical part and to guarantee that the foundations and the context of the project are correct, various sources and material were used such as articles, official documentation and previous studies regarding the optimization of online videogames, and that are being mentioned in the bibliography or cited correctly during the redaction of the document.

The coherence and structure is also being considered in order to write every section of this document, while also receiving feedback from Mónica Martín Mínguez, the director of this project to detect any possible conceptual errors, missing parts, suggestions or even redaction errors.

Additionally, tools such as Google Docs are used to, not only as a support to write the document, but to check for spelling and grammatical errors. The aim is to make this document more polished.

4.3. Practical Part Validation

To ensure that the objective of optimizing the network is working correctly, for each iteration, as mentioned earlier, will have a testing build with a certain feature in focus. Each of these builds will have a different outcome since they will have increased optimizations in comparison to previous iterations, so to compare with future or previous iterations, we need to capture the time response between server and client and see if there has been an improvement between iterations. The expected metric is a decrease in the response time.

The success of the network optimization will be conducted if the following benchmarks are correct:

- Latency: ensuring that the average latency stays below 50 ms for clients that are connected locally and under 150 ms for non local connections. At least for a connection within the same region.
- Synchronization accuracy: testing object replication and player stats synchronization with a maximum deviation of 10 ms.
- Scalability: ensuring stable performance equally for all the 4 players connected.

Additionally, playtesting will also be conducted at the end of each developing iteration, in which random players will test the game and see if the game experience is as fluid as it is trying to achieve.

To do so, each participant will have a build for the current iteration and will be notified before starting that the purpose of said build is to gather feedback, whether be improvements in terms of gameplay loop, latency or even reporting bugs that they might have encountered. This allows the developers to know what it is failing and see if it can be fixed on the next iteration.

5. Project Development

As mentioned in the objectives of this document, the project will focus on the creation of a cooperative multiplayer experience up to 4 players, in which the main genre is PvE (player vs environment). The main objective of this game is to offer a cooperative gameplay in which the players need to reach from an initial point to the final goal, while killing hordes of zombies while they are strategically managing their limited resources such as ammunition and healing items. The game's network architecture will follow a client-host model while using a peer-to-peer (P2P) transport layer. This means that the logic of the game will occur inside the player who is hosting the game, while the transportation of data will be from client to client.

Lights Out, which is going to be the name of this project, will begin in an initial room where players can break boxes to try their luck at obtaining items. Afterward, they must progress through the level by overcoming various obstacles, such as zombies or vehicles blocking the path. Once they reach the end of the level, which is finding the safedoor, the game ends.

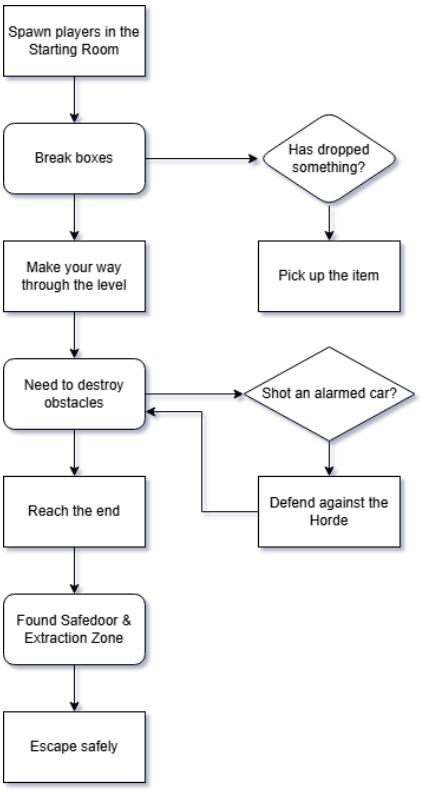


Figure 8: Gameplay flowchart

Players will be able to move, shoot, reload, heal themselves and jump, while the enemies will be controlled by an artificial intelligence to hinder the players.

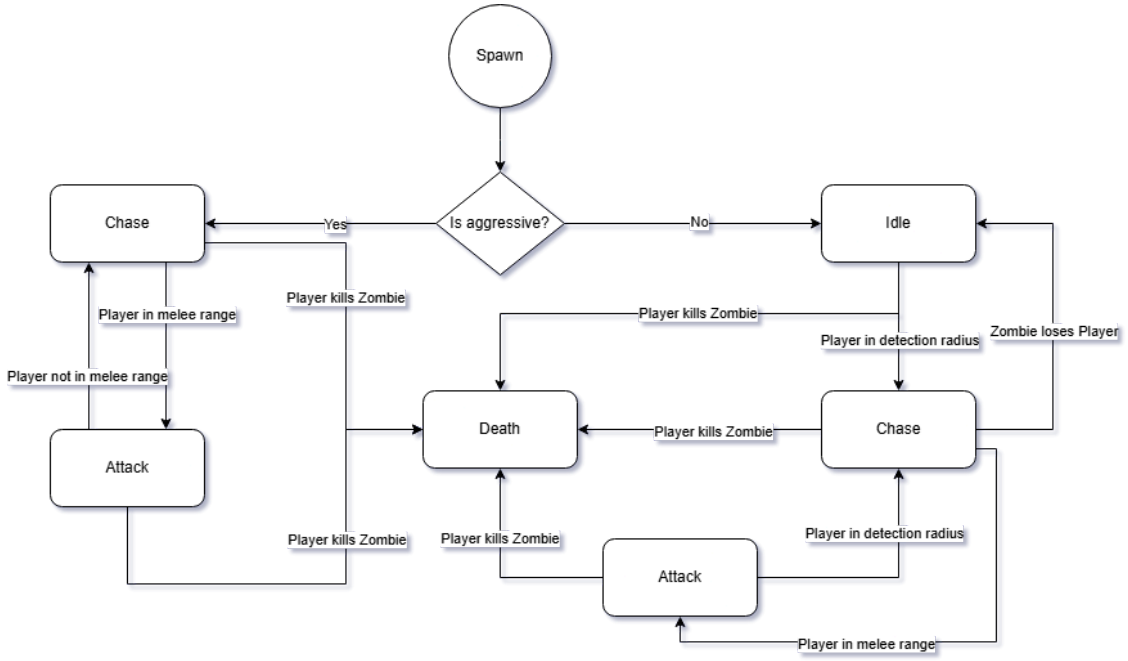


Figure 9: Zombie State machine diagram

Also, some of the features that it is intended to apply is the ability to add interactable and exploratory elements such as destroyable objects, that when shot, will break and reveal rewards such as ammo boxes, healing items or even reveal a hidden path. This type of mechanics also helps to enrich a better player experience by giving them the chance to explore the level.

And since the game will also feature procedural level generation, players can enjoy a different level each time they start a new game, where different paths can lead to the final goal of the level.

As for the flowchart of the videogame, an initial screen where it is going to detect whether the player is connected on Steam or not. If correct, redirects to the main menu, where the player can host a lobby, join a lobby through an lobby id, adjust volume settings and exit the game.
When inside a lobby, all the connected clients are able to use the chat to communicate with the other players. Only the host can start the game.

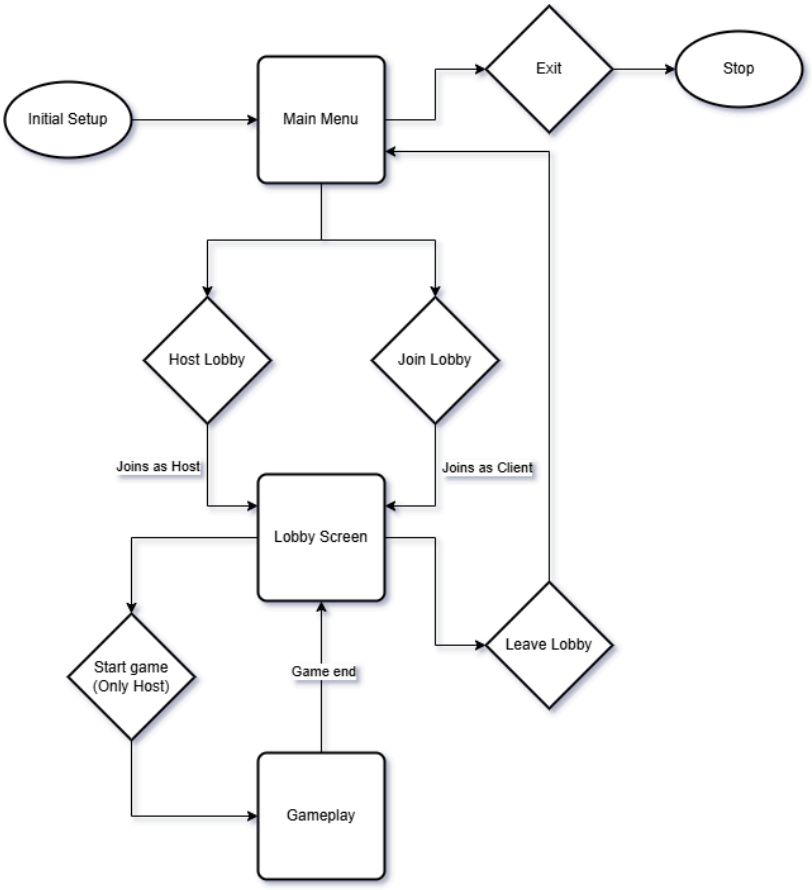


Figure 10: Lights Out flowchart diagram

This design is inspired by titles of reference, previously referenced in this document, like *Left 4 Dead*, *World War Z*, *Back 4 Blood* and *Remnant: From the Ashes* which have demonstrated and show high efficiency in terms of cooperation as a gameplay core in the zombie apocalypse genre. All of these games share a similar structure based on linear levels, as a difference that his project will feature procedural level generation (*Remnant: From the Ashes* do also feature procedural level generation)

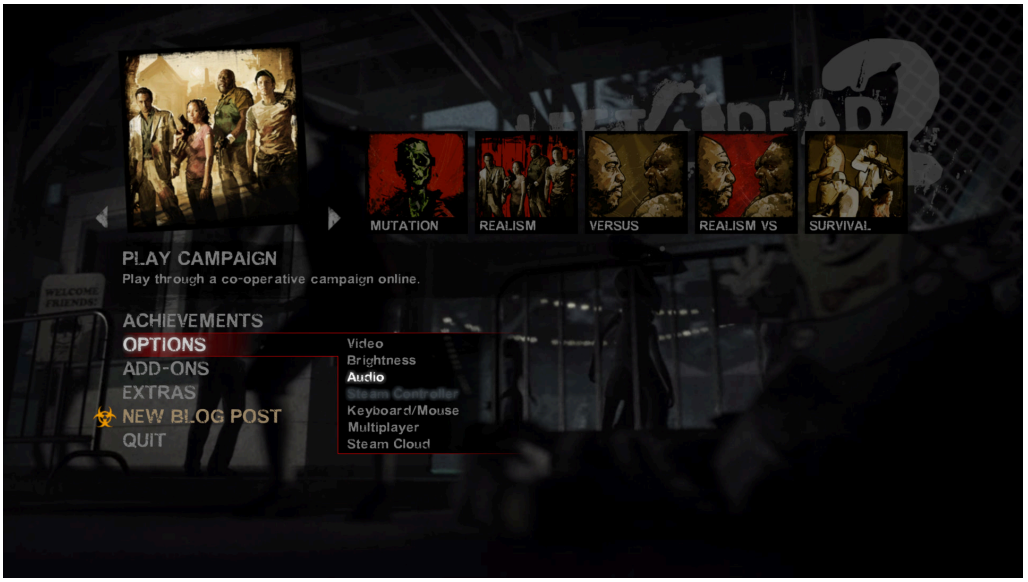


Figure 11: Left 4 Dead 2 main menu

In the point of view of the gameplay, various intended implementations are inspired by the essential actions that these videogames feature, such as the third person view that World War Z has. Also various actions that the players can do which is also frequent on all the zombie games is to shoot, heal and reload while being chased by the environment, in this case, by zombies. Although having a third person view is intended for this project, players will be able to change it to first person if they would like.

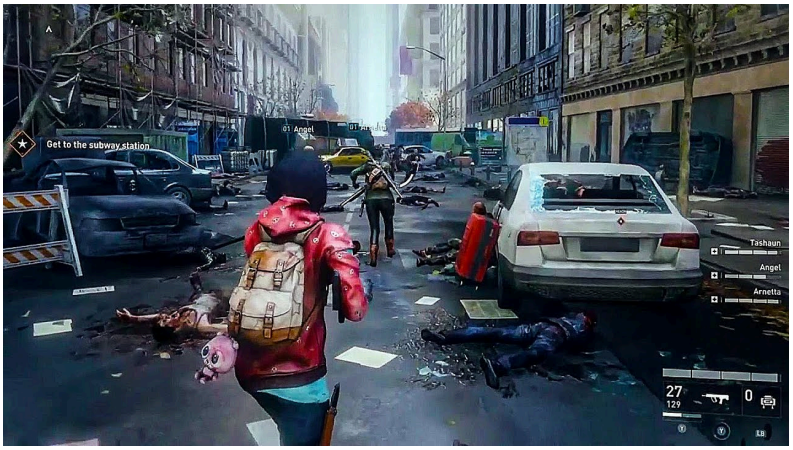


Figure 12: World War Z third person point of view

The main objective is to try to reach the safe room on the other point of the map and is also inspired by these titles. The safe room is, as its name implies, a safe room where no zombies can enter if the door is closed. This place is where the players can confirm the extraction point if reached safely.

Figure 13: Back 4 Blood safe room

As mentioned in the scope of the project section of this document, this project might feature missing animations, since the objective is to offer a multiplayer experience. Props will not be created from 0 but downloaded from the internet and are free to use as long as it is for educational purposes while not being for commercial purposes.

5.1. Iteration 1: Initial Planification

Following the initial planification, in this section, the basis of the project is being stated. By following the main objectives, the technologies that are going to be used are the next ones:

- Unity 2022.3.44f1 (LTS): the game engine that is going to be used is Unity. This version has been chosen since it is a LTS (long-term support) for its stability and continued updates.
- Custom netcode using C# sockets: instead of relying on existing network solutions, a fully custom networking done from zero is going to be implemented using Socket from .NET. This enables full control over the connection handling, data transmission and synchronization, targeted optimization for the specifics of the project and in-depth learning of networking.
- UDP-focused: multiplayer videogames are mostly handled via UDP due to its low latency, so for a videogame where the state is constantly updated each frame, sending the information fast is crucial.

Before starting the implementation of the basic core mechanics of the videogame, the custom netcode needs to be addressed as soon as possible since it is going to be the core and the one in charge of every single state of the game.

Developing a netcode system from scratch comes with greater complexity and time investment, but provides significant advantages:

- Enables precise adaptation of network traffic to the game's design requirements.
- Eliminates reliance on external libraries or restrictive integrations.
- It opens the door to future custom optimizations, such as packet compression, state prediction, and client reconciliation.

This part of the project is available in the following release as Lights Out v.0.3.

GameClient and GameServer architecture

As a consequence of deciding on a client-host architecture, where a computer acts as both server and client, and clients connect to that client-host, a *GameClient.cs* and *GameServer.cs* are needed to accomplish this.

GameServer.cs contains all the server-side logic of the network code. Its role is to receive and broadcast messages between all the connected clients while maintaining the synchronized state of all the networked objects and control game scene transitions. Some of the features that are implemented are the following:

- Singleton pattern: we need to ensure that only one instance of *GameServer* exists.
- UDP socket: receives data from the multiple clients connected to it. Using *SocketType.Dgram* and *ProtocolType.Udp* to define it.
- Message reception and processing: a separate thread, *mainReceivingThread()* handles the incoming data packets without blocking the game. These messages are decoded in a Queue to be processed during the *Update()* loop. Depending on which PacketType is received, a function will then be executed. These functions are specified and mapped in *StartServerFunctions()*.

```
C/C++
void StartServerFunctions()
{
    functionsDictionary = new Dictionary<PacketType, Action<object,
EndPoint>>()
    {
        { PacketType.Ping, (obj, ep) => { HandlePing(ep); } },
```

```
        { PacketType.Disconnect, (obj, ep) => { HandleDisconnect(ep); }
},
        { PacketType.PlayerData, (obj, ep) => { AddUserToDictionary(ep,
(Wrappers.UserData)obj); } },
        { PacketType.SceneLoadedFlag, (obj, ep) => {
HandleClientSceneLoaded(ep); } },
        { PacketType.playerActionsList, (obj, ep) => {
HandlePlayerActions((Wrappers.PlayerActionList)obj, ep); } },
    };
}
```

- Connected users: this script also keeps track of the current users connected in *connectedUsers* which also keeps track of each *EndPoint* and their associated username.
- Networked Object synchronization: server also keeps track of a *Dictionary*, *netObjectsInfo* with all the synchronized objects in the game. This information is periodically broadcasted among all the clients, including player actions by using *BroadCastPacket()*.

For the moment, the game only works on localhost, which is a special hostname that refers to the current computer that the game is being played on. It is essentially an alias for the loopback address, 127.0.0.1. This means that only the computers who are connected to the same network, in this case a router, can join the server who is hosting the game.

GameClient.cs contains the core logic for the multiplayer game client. The main purpose of this script is to manage the connection to the server, handling the sending and receiving of packet data, and update the networked objects in the game. Some of the features that are implemented are the following:

- Singleton pattern: like the *GameServer*, we need to ensure that only one instance of the client exists at all times.
- UDP connection: since the server uses a UDP socket, the client must also establish a UDP socket for the sending and receiving packets.
- Packet reception management: data is handled in a separate thread, *mainReceivingThread()*. Like the server, incoming packets are decoded and queued for processing on the main thread through a Queue. Depending on which PacketType is received, a function will then be executed. These functions are specified in *StartClientFunctions()*.

```
C/C++
void StartClientFunctions()
{
    functionsDictionary = new Dictionary<PacketType,Action<object>>() {
       { PacketType.Ping, obj => { HandlePing(); } },
       { PacketType.Disconnect, obj => { HandleDisconnect(); } },
       { PacketType.ChangeSceneCommand, obj => {
HandleSceneChange((Wrappers.ChangeSceneCommand)obj); } },
       { PacketType.netObjsDictionary, obj => {
HandleReceiveNetObjects((List<NetInfo>)obj); } },
       { PacketType.playerActionsList, obj => {
HandlePlayerActions((Wrappers.PlayerActionList)obj); } },
    };
}
```

- Ping and timeout system: client will periodically send a ping packet to the server to verify the connection. If no response is received within a certain timeframe, the client will automatically disconnect.
- Periodic player action sending: the player will eventually gather its actions and send them periodically to the server.
- Networked object synchronization: has a reference to the *NetObjectsHandler.cs* which is in charge of creating, updating, synchronizing and destroying networked objects.

Here is a comparison table between GameClient.cs and GameServer.cs from the project.

Aspect	GameClient	GameServer
Role	Connects to the server and receives game updates	Manages the game logic and synchronizes the state with all clients
Initialization	Connects to the server with a specified IP	Listens for connections in a specific port
Network management	Manages connection with the server and handles incoming data	Manages multiple client connections and handles data
Update	Sends player input to the server and updates local state based on server data	Processes inputs from clients and updates the game state
Synchronization	Receives and applies game state updates from the server	Sends authoritative game state updates to all connected clients

Table 9: GameClient and GameServer comparison table

PacketHandler

This script, crucial in a custom netcode, serves as the central hub for serializing and deserializing data packets exchanged between clients and server. The following features are implemented:

- Packet structures: specifies the format and types of packets that can be sent and received. This is done through an enum of *PacketType*.

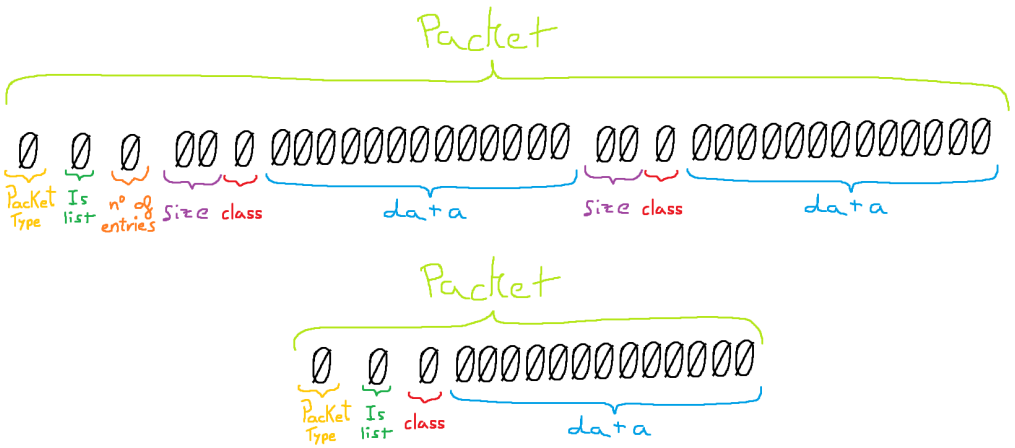


Figure 14: Packet structure

As shown in Figure 14, this is the packet structure defined for the project. The system accounts for two scenarios: when the packet contains a single message, and when it contains multiple messages grouped together. To distinguish between the two, the serialized data includes a byte indicating whether the stream consists of one or several packets. If it is a list, an additional value is included to specify the total number of packets contained within.

- Serialization and deserialization: includes methods to convert complex data structures into byte arrays for transmission (serialization) and reconstruct them back into usable objects upon reception (deserialization).

Both client and server scripts use the *PacketHandler* to manage their network communications. When a client or server needs to send information, serialization needs to be done in order to transmit it over the network. Upon receiving the data, the packet needs to be deserialized and depending on the type, a function will be then executed.

Also, we need to overload the *SendPacket()* function, so that whenever we need to send a single packet or multiple packets, we add as an argument a *List<NetInfo>* or a single *NetInfo*. In the *Wrappers* section, *NetInfo* is being defined.

Wrappers

Wrappers are a collection of serializable data transfer classes that are used to send and/or receive game object information over the network. They serve with the sole purpose of sending object data information into a transmittable format that supports polymorphic objects. For example, those GameObjects that extend from *NetInfo*, will be able to use the *Wrapper* data to update its internal values.

NetInfo is an interface created for the purpose of adding Serialize() and Deserialize() functions so that it can be handled by the PacketHandler.

An example of a *NetInfo*, will be the *UserData*, which includes the serialize and deserialize functions of the username. Here's the implementation:

```
C/C++
[Serializable]
public struct UserData : NetInfo
{
    public string userName;

    public UserData(string username)
    {
        userName = username;
    }

    public byte[] Serialize()
    {
        MemoryStream stream = new MemoryStream();
        BinaryWriter writer = new BinaryWriter(stream);

        writer.Write(userName);

        byte[] data = stream.ToArray();

        stream.Close();
        writer.Close();

        return data;
    }

    public void Deserialize(byte[] data)
    {
        MemoryStream stream = new MemoryStream(data);
        BinaryReader reader = new BinaryReader(stream);

        userName = reader.ReadString();

        stream.Close();
```

```
        }
    }
```

Notice that depending on the type of variable that it is trying to read, it needs to call the corresponding function from the BinaryReader class to read the type of variable. And depending on the order of serializing, the deserializing part needs to be done with the same order.

NetObjectsHandler

This script is in charge of instantiating the networked objects, such as players, enemies, etc, when arriving at new data coming from the server. It also updates their transform component (position, rotation) and their state, which also includes their destroy management.

Its structure contains a prefab mapping, which relates the type of objects from the *Wrappers* to a predefined prefab:

```
C/C++
{ typeof(Wrappers.Player), "PlayerPrefab" },
{ typeof(Wrappers.BasicZombie), "BasicEnemyPrefab" },
```

There are several functions that manage the destruction of the objects when a packet containing information of the networked objects needs to be deleted on all clients. For example, killing a zombie requires all clients to destroy that game object so that the game state is synchronized among all peers connected.

Action-Based Input System

In order to send player inputs through the network, it is not efficient to send directly the position, which is a vector of 3 floats, a rotation which is a vector of 4 floats. To optimize the amount of data sent, instead of sending these variables, we are sending the type of action that the player has made. In *Wrappers*, there is the definition of the different actions that the player can do:

```
C/C++
public enum ActionType
{
    None,
    MoveF,
```

```
    MoveB,
    MoveL,
    MoveR,
    Rotate,
    Shot,
    OpenDoor,
}
```

Each of these enums are represented like an int, which size is 4 bytes, with this, the optimization of sending many variables for the position (3 floats) and the rotation (4 floats), which sums up to 28 bytes in total, we are saving 24 bytes worth of space.

All of these implementations can be found in the following repository, which can also be found in the Links section of the document.

Aspect	Raw Transform Data	ActionType enum
What is sent	Position (Vector3 = 3 floats) + Rotation (Quaternion = 4 floats)	A single enum value representing the player's action
Data size	3 floats (12 bytes) + 4 floats (16 bytes) = 28 bytes	1 integer (4 bytes) = 4 bytes
Structure	Unity's built-in Vector3 and Quaternion	Custom ActionType enum
Efficiency	High bandwidth usage per frame	Significantly reduced bandwidth usage

Table 10: Raw Transform Data vs ActionType enum

Player & Zombie

For this implementation of the player behavior, a simple movement and camera control and shooting with the left mouse click will be implemented.

For the zombie, only movement is implemented and when getting killed, marked to be deleted in the server.

But since it needs to be integrated with the custom netcode, the following specifications were needed to be considered:

- Authority: determining if the current instance has the authority to control the player, which avoids synchronization problems (*isOwner*).
- Position and rotation: position is updated through the *PlayerActions enum*, which is sent to the server, acknowledging it and sending back to the other clients. For those

entities who have the same script, will have their transform updated by receiving the *PlayerActions enum* and the *deltaTime* of the client who sent it.

```
C/C++
void Move(PlayerAction.ActionType action, float deltaTime)
{
    Vector3 direction = Vector3.zero;

    switch (action)
    {
        case PlayerAction.ActionType.MoveF:
            direction = transform.forward;
            break;
        case PlayerAction.ActionType.MoveB:
            direction = -transform.forward;
            break;
        case PlayerAction.ActionType.MoveL:
            direction = -transform.right;
            break;
        case PlayerAction.ActionType.MoveR:
            direction = transform.right;
            break;
        case PlayerAction.ActionType.None:
        default:
            break;
    }

    if (direction.magnitude > 0)
    {
        transform.Translate(direction * moveSpeed * deltaTime,
Space.World);

        if (transform.position != lastPosition) positionChanged = true;

    }
}
```

- Gun: to send where the bullet goes, the client sends the information of the trail created when shooting the gun.
- TakeDamage: for the zombie, when hit enough, it will be marked as to be deleted and will then send the information to the server. Then, the server acknowledges it and sends back a packet with the information of the updated net objects, so that the clients destroy it safely.
- NetObject: since it needs to be sent to the network, it requires to be serialized and deserialized, as mentioned in the *NetObjectsHandler* section.

Iteration testing

The game has been tested out with 22 people. Overall, their feedback regarding the 0.3 version of the game is that the person who is hosting the game is not experiencing lag issues (which is

expected since he is the one doing all the logic of the game on his local side) and those who are connected to the host are experiencing warping, which is a phenomenon that when the user wants to move, let's say for example, forward, the game teleports him back to the previous position, making this a undesirable experience.

5.2. Iteration 1.5: Relay Server

For this iteration, the idea is that we have to replace the localhost implementation to a Relay Server. A Relay Server is an intermediary server that receives the packets from all the clients and the host, and rebroadcasts them to the intended receiver. This is a solution for those multiplayer games that need a certain port to be opened on their router, but since cloud servers do have all the ports opened, the NAT traversal (or NAT punchthrough) which is a technique that enables communication between devices on private networks, is no longer needed, which makes this a good solution for communicating between peers.

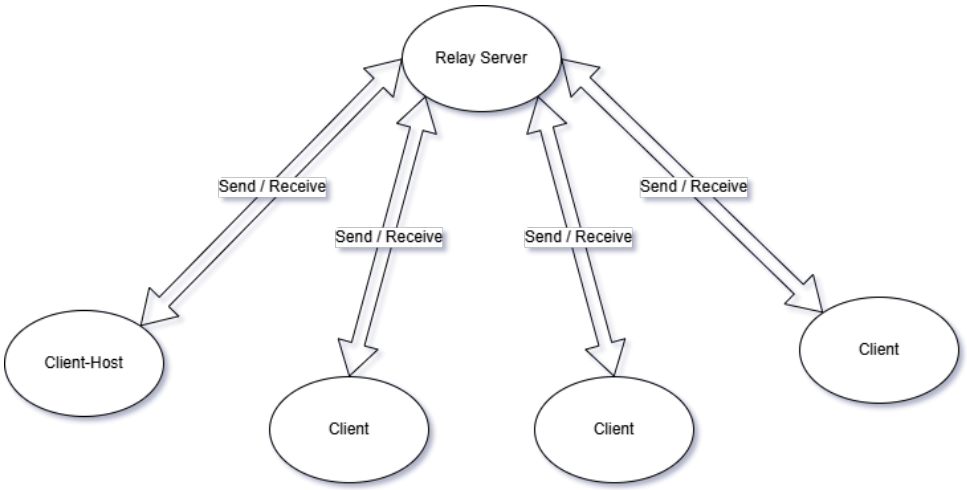


Figure 15: Relay Server diagram

In order to implement this, we need a cloud server that will act as the Relay Server. For this implementation, Amazon Web Services will be used in order to achieve this behavior. We need to launch an EC2 virtual machine with an operating system that Unity is compatible with and apply port forwarding to the port that it is going to be used for. Pòrt forwarding is redirecting network traffic from one port of a router to another. In this case it is a router to a server.

For the project, the port 9050 with the UDP protocol is the one it is going to be used for and the operating system chosen is Linux.

Once the server is configured, we are ready to implement the relay server in Unity, in which we are going to create a headless application. A headless application is an executable program with no interface, which only needs a console to register logs. Since we are using Linux as our operating system in the cloud server, a package containing Linux builders is needed to compile a build so that it can be uploaded to the server.

RelayServer.cs features almost the same as *GameServer.cs* but with slight differences:

```
C/C++
void Receive()
{
    IPEndPoint sender = new IPEndPoint(IPAddress.Any, 0);
    EndPoint Remote = (EndPoint)(sender);
    byte[] data = new byte[1024];
    int recv;

    while (true)
    {
        try
        {
            recv = relaySocket.ReceiveFrom(data, ref Remote);
            if (recv == 0) { continue; }
        }
        catch (SocketException ex)
        {
            Debug.LogWarning("SocketException error (RelayServer
Receive()): " + ex.Message);
        }

        Debug.Log($"New client connected: {Remote}");

        // Add IP if it does not exist
        if (!connectedIps.Contains(sender)) connectedIps.Add(sender);

        (PacketType, object) decodedClass;

        // Data[1] defines if the packet contains a list
        // 0 - Single packet
        // 1 - Multi packet
        if (data[1] != 0) decodedClass =
PacketHandler.DecodeMultiPacket(data);
        else decodedClass = PacketHandler.DecodeSinglePacket(data);


        if (decodedClass.Item1.Equals(PacketType.Disconnect))
        {
```

```
            functionsQueue.Enqueue((decodedClass.Item1,
decodedClass.Item2, Remote));
        }

        // Re send packet to all clients
        foreach (IPEndPoint client in connectedIps)
        {
            relaySocket.SendTo(data, data.Length, SocketFlags.None,
client);
        }
    }
}
```

From the script, the only difference is that it re-sends the packet among all the connected clients, including the server but except the client who has sent the packet. Also has their own mapped function depending on the *PacketType* received and it is executed on the main thread.

Now that the script is done, uploading it to the cloud server is necessary to make it work. For that purpose, the cloud server must be initialized in the Amazon Web Service control panel and generate the .pem file, which is a private key to access the cloud server. With this file, WinSCP, which is a software that allows the user to upload files in servers, is going to be used in order to access the server and the private IP. Once that is done, create the build in Unity, with the option “Dedicated Server” in the Linux section. This process will then create a headless application, which is an app that when executed will not feature a graphical user interface, but will run in the background.

Then, uploading the build and executing the file with the following commands:

```
C/C++
chmod +x RelayServer.x86_64
./RelayServer.x86_64
```

The relay server will be ready to receive and resend the packets received from the different clients, and to make sure it is doing it, in the game the public IP of the server needs to be used instead of the localhost mentioned earlier.

For this iteration, sending a packet through the network so that the server receives it works correctly, but the problem occurs when retransmitting the packet to the other peers does not work as intended. Having full control of the custom netcode might have its advantages, but

challenges the developer to handle every single detail. For this reason, the idea of having a custom code is discarded, currently abandoning the idea of using a custom relay server.

5.3. Iteration 2: Project Rework, NGO & Steamworks

For this iteration, it has been decided with my project director a change of objectives. A custom code will no longer be developed but instead it will be using Netcode for GameObjects (NGO), a Unity library that handles network connectivity, including remote procedure calls (RPCs) and intended for client-host architecture. The objectives are already updated in the objectives section of the document.

Also, Amazon Web Services is no longer to be used for this project, and decided to use Steamworks API, which handles the NAT traversal smoothly and uses the Steam platform functionalities, which includes creating player lobbies, Steam achievements, Steam friends, etc. A whole rework is needed, because Sockets are no longer part of the project and it needs to be adapted to NGO. That is why a new repository, which can also be found on the Links section, has been created to make this transition smoothly.

Initial setup

For starting with NGO and Steamworks, we need to install a network transport that connects the Unity game to the Steamworks API. The files that are required to be installed can be found in the following repository, and only get the Facepunch transport. By dragging the Facepunch folder inside the Assets folder of the project, the network transport is installed.

This part of the project is available in the following release as Lights Out v.0.4.

So, for the initial setup, it is important that in Unity we have a GameObject with the *NetworkManager.cs* script that the package gives, which does all the network functionality from the previous iterations, but prepared for production purposes. This script has a singleton pattern in which as mentioned earlier, it is needed so that it can only be one instance of it. And since the Facepunch transport has been installed, changing the network transport to it is needed to use the Steamworks API.

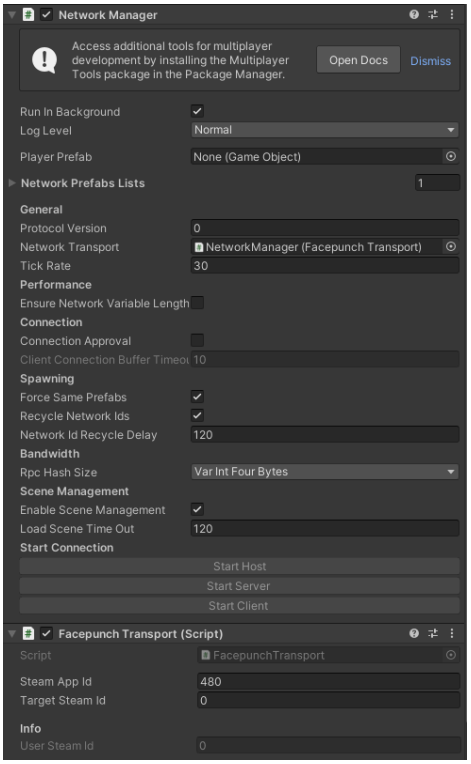


Figure 16: NetworkManager with Facepunch Transport

Notice that the Steam App Id in the Facepunch Transport component, it is set as 480, which is the App ID that Steam provides to developers to test out their videogames during production phase. It is mandatory that the computer where the game is being developed to have the Steam platform application opened so that the connections work well. In case this does not work, the developer will require a copy of *Spacewar!*, which is the Steam app whose App ID 480, that serves as the placeholder before publishing a game in the store. For those who do not have *Spacewar!*, anyone with a Steam account can write in the web browser

```
C/C++
steam://install/480/
```

If Steam is installed, it will automatically launch and prompt the user to download the game. That confirms that *Spacewar!* has been registered correctly on the Steam account. If no prompt appears, search in the library for the game. If it is there, you are ready to go.

To verify that the *NetworkManager* integrates correctly with Steamworks API, add an empty GameObject with the *NetworkManager* script and attach the following script with a Coroutine:

```
C/C++
IEnumerator LoadMainmenuScene()
{
    yield return new WaitUntil(() => NetworkManager.Singleton != null);
    SceneManager.LoadScene("1_MainMenu");
}
```

If the scene loads successfully, it indicates that the connection towards the Steam API is working correctly.

Lobbies

Lobbies were not implemented in the previous iterations, but with the Steamworks API, this task should be feasible. For this purpose, a simple UI needs to be implemented to show on screen the Lobby ID and to allow users to input said ID to join a Lobby.

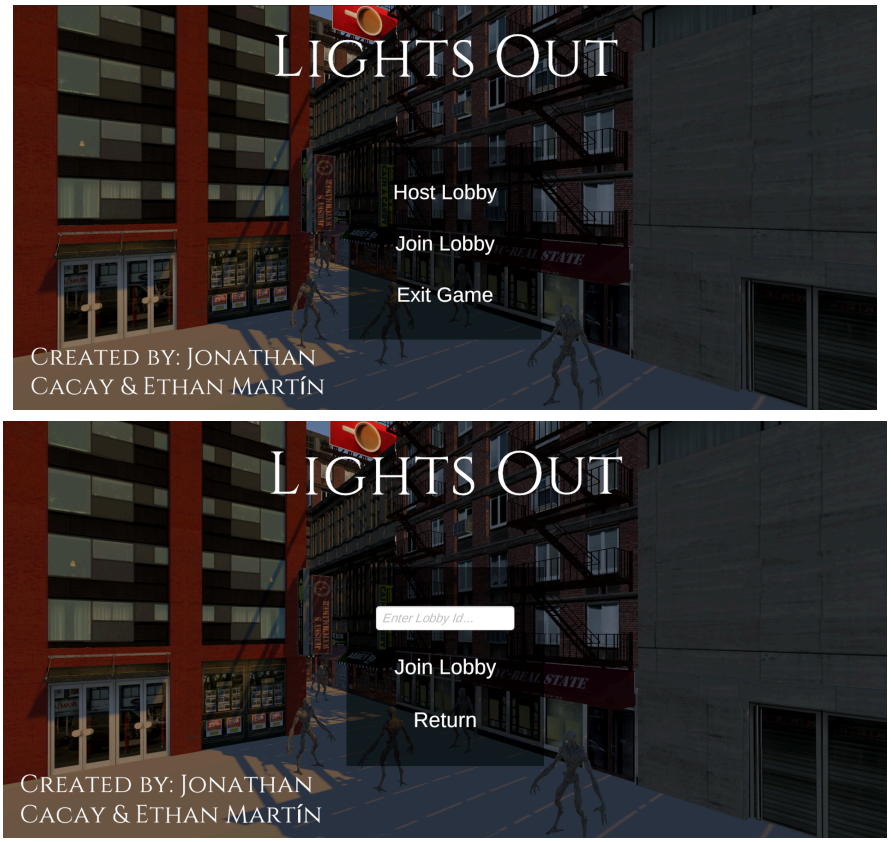


Figure 17: Lights Out initial User Interface

SteamManager.cs has been created to handle everything that requires to use the Steamworks API. To host a lobby, is as simple as calling the following function:

```
C/C++
public async void HostLobby()
{
    await SteamMatchmaking.CreateLobbyAsync(maxPlayers);
}
```

Notice that this is an asynchronous call, which is a function that does not have an immediate response, that is why we need to add the *await* keyword so that it waits until a Lobby is created with a given number of players.

To make the join Lobby functionality with a given Lobby ID, the following function has been implemented:

```
C/C++
public async void JoinLobbyWithID()
{
    ulong ID;

    // Steam lobby IDs are 'ulong' type of variable
    if (!ulong.TryParse(LobbyIDInputField.text, out ID))
    {
        Debug.Log($"{LobbyIDInputField.text} is not a valid Lobby ID.");
        PlayerFeedback.text = $"{LobbyIDInputField.text} is not a valid
Lobby ID.";
        return;
    }

    try
    {
        Lobby l = new Lobby(ID);
        await l.Join();
    }
    catch (Exception ex)
    {
        Debug.Log($"Failed to join lobby: {ID}, {ex.Message}");
        PlayerFeedback.text = $"Failed to join lobby: {ID},
{ex.Message}";
    }

}
```

From the script, we make sure that the ID that the player has input is at least an ulong (unsigned long) since Lobby IDs are from that type. If it is not, we inform the player that the ID is not correct. If it is correct, we then attempt to join the Lobby with the associated ID.

To facilitate copying the lobby ID, a button has been added to the lobby screen so that whenever the button is pressed, the ID is copied in the clipboard, as if the user has executed CTRL + C for copying something.

To make sure that changes to joining or leaving a lobby are well implemented, we need to make sure that certain functions are subscribed to custom made functions. We can subscribe to them when enabling the script.

```
C/C++
void OnEnable()
{
    SteamMatchmaking.OnLobbyCreated += LobbyCreated;
    SteamMatchmaking.OnLobbyEntered += LobbyEntered;
    SteamFriends.OnGameLobbyJoinRequested += GameLobbyJoinRequested;
    SteamMatchmaking.OnLobbyMemberJoined += LobbyMemberJoined;
    SteamMatchmaking.OnLobbyMemberLeave += LobbyMemberLeave;
}
```

Then, once that is done, we can tailor the desired behaviors when creating, entering or requesting to join a lobby.

Player list

In a multiplayer videogame, especially before starting the game, it is important to know who the players are in the lobby. This information helps to establish clarity, gives a sense of presence and ensures that players are aware of who they are about to play with, by displaying the name and their profile image.

By using the functions that Steamworks offers, a function has been created to help showcasing that information.

```
C/C++
async void UpdateUI()
{
    foreach (var player in currentPlayers)
    {
        Destroy(player);
    }
    currentPlayers.Clear();

    foreach (var player in
LobbyReference.Singleton.currentLobby?.Members)
    {
        GameObject playerItem = Instantiate(playerInfoPrefab,
LobbyPlayersList.transform);
        PlayerInfoUI playerInfo =
playerItem.GetComponentInChildren<PlayerInfoUI>();

        playerInfo.playerName.text = player.Name;
```

```
        Steamworks.Data.Image? image = await
player.GetLargeAvatarAsync();

        if (image != null)
        {
            Texture2D tex2d = new Texture2D((int)image.Value.Width,
(int)image.Value.Height, TextureFormat.RGBA32, false);
            tex2d.LoadRawTextureData(image.Value.Data);
            tex2d.Apply();

            playerInfo.playerImage.texture = tex2d;
        }

        currentPlayers.Add(playerItem);
    }

    Canvas.ForceUpdateCanvases();

LayoutRebuilder.ForceRebuildLayoutImmediate(LobbyPlayersList.GetComponen
t<RectTransform>());
}
```

From the script, we gather the information of all the connected players in the Lobby and attach them to a predefined Prefab, *playerInfoPrefab*, and added as a child of the lobby player list container, *LobbyPlayersList*. For the user name, it is straightforward, by calling *Friend.Name* and setting it to the corresponding UI component, but for the avatar image, we need to do an asynchronous call to get the image information from the Steam account. If it is not encountered, the default image from the Prefab will be loaded, if it is not *null*, then the image from the Steam account will be displayed.

Figure 18: List of players with only 1 player

Lobby chat

Communication between players before a match starts is also key to coordination, engagement and a smooth user experience. A basic lobby chat where users interact with each other in real time, whether to discuss strategy, socialize or confirming readiness. Adding a lobby chat helps create a more interactive experience and environment in a cooperative multiplayer videogame. Steamworks API enables a simple but effective real-time chat that automatically logs player events such as join and leave.

To implement this, like the previous implementation, a UI is needed to show the chat log. Then registering the necessary events from the Steamworks API to the custom made functions made for the chat events.

So, how do chat messages work in the Steamworks API?

- Every single lobby has a channel for chat. When creating or joining a Steam Lobby, it automatically gives access to a chat system given by the Steam platform that allows sending messages to all the connected peers. All is needed is to make the callbacks to the custom made functions.

```
C/C++
void OnEnable()
{
    SteamMatchmaking.OnChatMessage += ChatSent;
    SteamMatchmaking.OnLobbyEntered += LobbyEntered;
    SteamMatchmaking.OnLobbyMemberJoined += LobbyMemberJoined;
    SteamMatchmaking.OnLobbyMemberLeave += LobbyMemberLeave;
    ChatGameObject.GetComponent<TextMeshProUGUI>().text = "";
}
```

- Sending a message through the API converts the message into a data packet that Steam broadcasts among all the connected players in the Lobby. Internally, Steam serializes the string and broadcasts it to all the clients.

```
C/C++
void SendMessage()
{
    if (!string.IsNullOrEmpty(ChatInputField.text))
    {

LobbyReference.Singleton.currentLobby?.SendChatString(ChatInputField.tex
t);
        ChatInputField.text = "";
    }
```

```
}
```

- To receive messages it is done through the *SteamMatchmaking.OnChatMessage* event which triggers whenever anyone in the lobby, including yourself, sends a message.

The only inconvenience that Steamworks API has is that the size of the messages need to be no longer than 256 characters.

Iteration testing

For this iteration, a total of 6 people were helping out in order to give feedback. The objective of this testing is to see if the Lobby functionalities were working properly within the Steam context.

During the sessions, I was constantly communicating with each of the testers who were testing out the release. Whenever they were interacting with the Lobby functionalities, I was asking them directly regarding their first impressions or if they were missing something that usually a multiplayer game offers.

Overall, what they liked most about is the capacity of using a text chat, so that they could communicate with the clients connected in the lobby. One thing they did miss is having the possibility to kick clients who are the host of the lobby.

Something negative they told me is that they needed to have to use *Spacewar!* as an intermediary for the game was bad, since it is seen normally as a pirated game on Steam, but that is not the purpose of that game, but to help developers use it as a placeholder before publishing the game in the Steam store. Spacewar! Is more like a technical test or system to authenticate Steam users, oftenly used in the production phase of a multiplayer game destined to be published on Steam.

5.4. Iteration 3: Improving Game Loop functionalities

As a part of this iteration, it is necessary to adapt the scripts of the Player and the Zombie behaviors, which were implemented in a custom netcode based on C# Sockets to Netcode for GameObjects (NGO). This means that the logic intended on the mentioned scripts needs to be reworked to adjust the NGO components, in which they need to use *NetworkBehaviour*-derived functions and components, such as *NetworkTransform* (a variable

to update position and rotation through the network) and *NetworkVariable* (a variable of any type that can be updated through the network).

Also, remote procedure calls (RPC's) are needed to be implemented using NGO's *ServerRPC* and *ClientRPC* methods to apply the network layer that it offers.

This part of the project is available in the following release as Lights Out v.0.5.

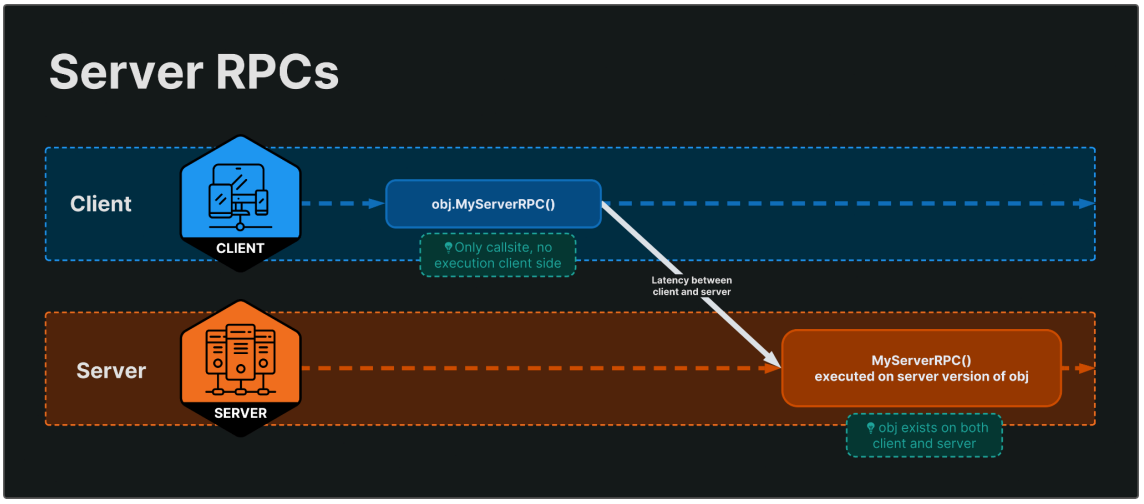


Figure 19: Server RPCs diagram

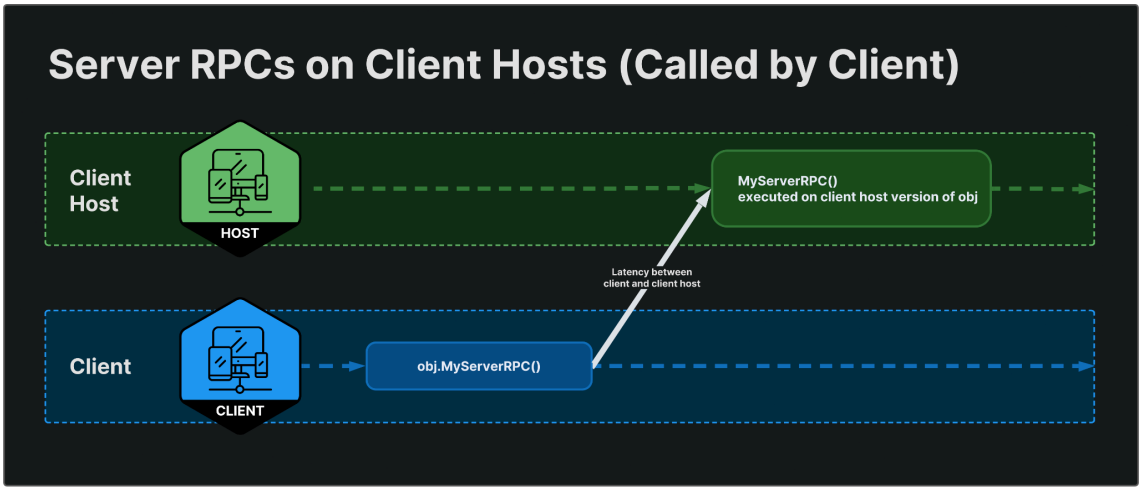


Figure 20: Server RPCs diagram on Client Hosts

As seen from the figures above from the Netcode for GameObjects online documentation, a client can invoke ServerRpc on a *NetworkObject* (those objects that need to be synchronized among all the connected clients). The RPC will be placed in the local queue and then sent to the server at the end of the frame. Upon receiving the server RPC, it is executed on the Server's instance on the same *NetworkObject*.

Also, during the process of development, we have to take into account that when running a game as a host, NGO invokes RPCs immediately within the same stack as the method invoking the RPC. Since a host is both considered a server and a client, it is recommended to avoid design patterns where a *ClientRpc* invokes a *ServerRpc* that also calls the same *ClientRpc* as it can lead to a stack overflow.

Player

As mentioned above, the player behavior needs to be reworked since it is using C# Sockets. So for this purpose. Many of the functions that were using Sockets were deleted for this iteration and needed an adaptation for NGO.

For example, moving the player character no longers uses the *Wrappers* custom containers, and the functions that were using it were modified as follows:

```
C/C++
[ServerRpc]
void SubmitMoveServerRpc(PlayerAction actionType)
{
	Move(actionType, Time.deltaTime);
}

[ServerRpc]
void SubmitRotateServerRpc(float mouseX, float mouseY)
{
	Rotate(mouseX * Time.deltaTime, mouseY * Time.deltaTime);
}

[ServerRpc]
void SubmitShotServerRpc(Vector3 origin, Vector3 dir)
{
	currentGun.Shoot(origin, dir);
}
```

Whenever a movement, rotation or gameplay action such as shooting the gun needs to be synchronized across all the connected clients, a *ServerRpc* is invoked, allowing all clients to request the server to perform an action, which can then be propagated to all other clients through *ClientRpc* calls.

Compared to previous iterations, several new improvements have been introduced. For instance, the default gun now includes not only aesthetic enhancements but also functional features such as giving the ability to reload it when the ammunition is empty or when the

player requires it. For this iteration, this is more than enough. In future iterations it is planned to add limited ammunition and the need of looting ammo boxes around the level map.

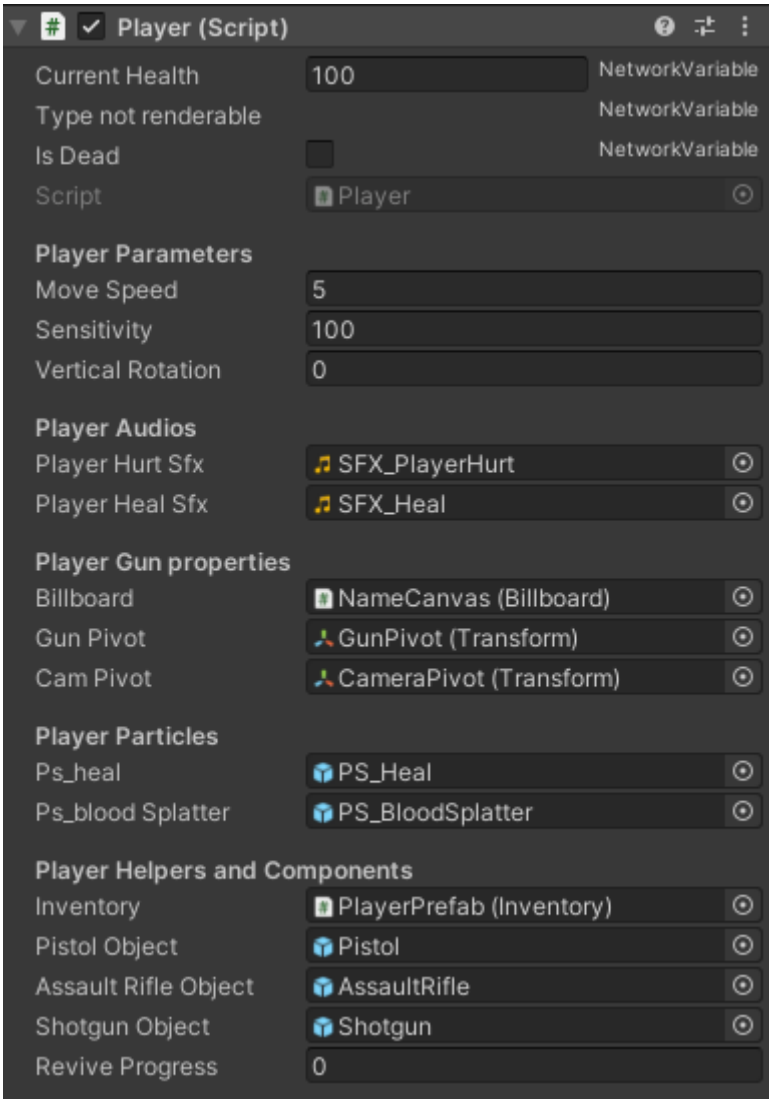


Figure 21: Player script component

Zombie

Like the Player, the Zombie behavior has also undergone several changes to be adapted in the new implementation. For instance, Zombies have *NetworkVariables* that need to be updated every time a player shoots them, such as *currentHealth*. Also having track of its current State so that other players can see the updated state that they have.

Since we are talking about a *NetworkObject* that needs to be updated, but it is not controlled by any player but the server itself, we need to implement *OnNetworkSpawn()*, which is called whenever a *NetworkObject* is spawned throughout the game.

Some of the changes that Zombie has undergone are the following:

```
C/C++
[ServerRpc]
public void TakeDamageServerRpc(int amount)
{
if (currentHealth.Value <= 0 || isDead) return;

currentHealth.Value -= amount;

if (currentHealth.Value <= 0)
{
   isDead = true;
   if (!CheckAnimationState("Death"))
  zombieAnimator.SetTrigger("Death");

        agent.ResetPath();
        audioSource.PlayOneShot(zombieDeathSfx);

        StartCoroutine(WaitForDeathAnimation());
    }
}
```

Like the Player, since this function was needed to be called when a Zombie received damage, it also needed to be adapted into a *ServerRpc* call, so that every single client can have the same numbers as the others.

As an addition from previous iterations, Zombie could not attack the player. That is why for this iteration the *DoMelee()* function has been implemented which triggers the animation and spawns a hurt collider in front of the Zombie.

This is handled as follows:

```
C/C++
IEnumerator SpawnMeleeHitbox()
{
    isAttacking = true;

    GameObject currentHitbox = Instantiate(meleeHitboxPrefab,
meleeSpawnpoint.position, meleeSpawnpoint.rotation);
    currentHitbox.GetComponent<NetworkObject>().Spawn();

    currentHitbox.GetComponent<ZombieDamageHitbox>().attacker = this;

    AnimatorStateInfo animatorStateInfo =
zombieAnimator.GetCurrentAnimatorStateInfo(0);

    while (!animatorStateInfo.IsName("Attack1"))
    {
```

```
        yield return null;
        animatorStateInfo =
zombieAnimator.GetCurrentAnimatorStateInfo(0);
    }

    yield return new WaitForSeconds(animatorStateInfo.length);

    if (currentHitbox != null)
    {
        if (currentHitbox.TryGetComponent(out NetworkObject networkObj))
        {
            networkObj.Despawn();
        }
        else
        {
            Destroy(currentHitbox);
        }
    }
    isAttacking = false;
}

// ZombieDamageHitbox.cs

public class ZombieDamageHitbox : MonoBehaviour
{
       public BasicZombie attacker;

       void OnTriggerEnter(Collider other)
       {
           if (other.CompareTag("Player"))
           {
               if (attacker != null)
               {
       other.GetComponent<Player>().TakeDamageServerRpc(attacker.attackD
       amage);
                   Debug.Log("Player hit!");
               }
           }
       }
}
```

Notice here that the collider calls the Player's *TakeDamageServerRpc()* function to synchronize the data. With this, when calling the function it ensures that the damage event is registered on the server and then properly synchronized with all clients.

Figure 22: Zombie script component

Extraction point

The extraction point, which is the green zone where all the players need to stand, is the main objective of the game: reach it at all costs. This entity is just an entry collider detects when a Player enters it or when it leaves it. If all the Players inside the game have entered the collider and stays inside it, then the game will finish and move all the players to the Lobby scene.

In addition, a *LevelManager* was also created to manage control of what is happening inside the level that the players are in. For the moment, it only has that behavior. In later iterations, it is intended to implement an *EntityManager* where the *LevelManager* will handle when a horde needs to spawn, when a zombie needs to spawn, etc.

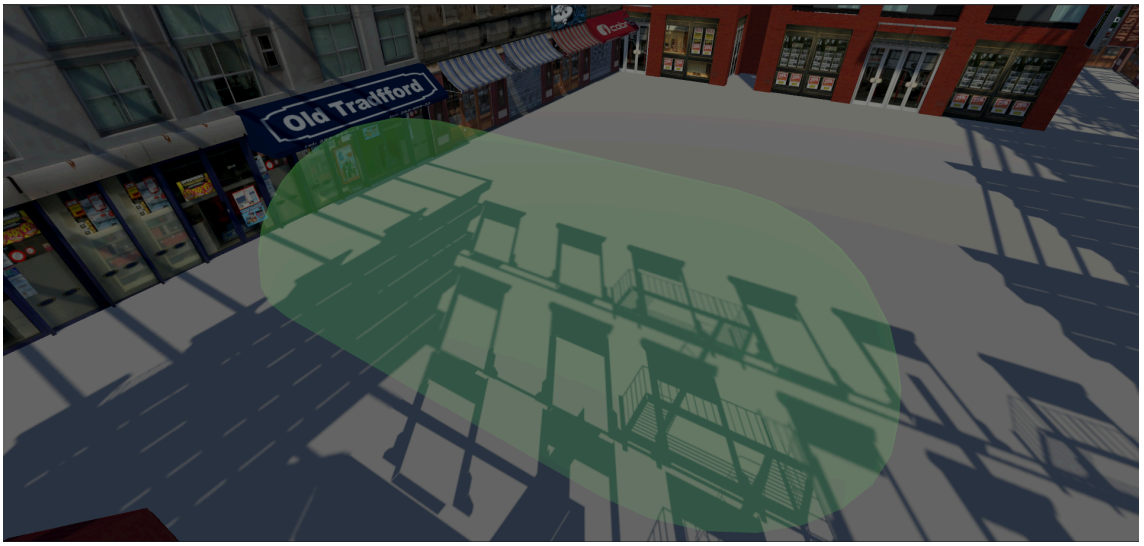


Figure 23: Extraction point

For future iterations, it is intended to improve this collider into a safe room in which players need to open the door, make all the players alive inside the safe room and then close the door to confirm the extraction.

Pick Up Items

A pick up item is an entity that a player needs to pass through to obtain it and use it to its benefit. These items can be new weapon variations or medkits to heal the player whenever it wants. For the weapon variations, it will unlock the weapon variation for the player who picks it up.

Figure 24: Pick Up Items

Iteration testing

For this iteration, the developers have tested each of the new implementations, to verify whether the behaviors from the zombies and player are as intended. Overall, this testing was needed to ensure that the sockets implementation from an early version was properly adapted to Netcode for GameObjects as an addition of the new functionalities implemented.

Overall, the implementation was working well, but it needed some adjustments such as optimization and some actions not being sent properly.

5.5. Iteration 4: Additional implementation & Bug Fixing

Now that the base game is adapted to the new framework, adding new features to the game should be easier; for this iteration, it is intended to implement zombie variations, boxes with random drops, better zombie feedback when hitting them, a simple UI, safedoor, point system and ranking, smooth movement in position and rotation, sounds synchronization, etc.

This part of the project is available in the following releases as Lights Out v.0.6 alongside Lights Out v.0.7.

Smooth movement

When a player client attempts to move the player character or rotate the camera view, a slight delay can be noticed making it unbearable for those players who are not the host. In a multiplayer videogame this responsiveness must be almost immediate to offer an equal smooth experience and not be an exception only for the host.

To address this problem, the previous solution done in Iteration 3, which was done through RPCs, is no longer used. An initial solution was to implement an input prediction algorithm that tries to assume a certain movement from the player. With this idea in mind, whenever a player client, which is owner of its own characters, attempts to move, it will try to move immediately in the direction it attempted on the client's point of view, then send it to the player host to acknowledge this movement and send it back to all the clients including the player who sent it so that every player can see the same movements. With this idea, the movement had a slight delay, better than earlier but still not enough.

```C/C++
public float moveSpeed = 5.0f;
Vector3 networkPosition;

void Update()
{
    if (IsOwner)
    {
        LocalMoving();
        SentPositionToServerRpc(transform.position);
    }
    else
    {
        transform.position = networkPosition;
    }
}

void LocalMoving()
{
    float horizontalInput = Input.GetAxis("Horizontal");
    float verticalInput = Input.GetAxis("Vertical");

    Vector3 movement = new Vector3(horizontalInput, verticalInput);
    transform.Translate(movement * speed * Time.deltaTime);
}

[ServerRpc]
void SentPositionToServerRpc(Vector3 position)
{
    SentPositionFromClientRpc(position);
}

[ClientRpc]
void SentPositionFromClientRpc(Vector3 position)
{
    if (IsOwner) return;

    networkPosition = position;
}
```

But this approach was not working as intended, so it was discarded completely from the Player implementation.

Reading the documentation from *Netcode for GameObjects*, *ClientNetworkTransform* was mentioned which fixes the problem of latency that was mentioned earlier; it offers a balance of between synchronized transforms and the latency between applying the updates on all the connected clients. Basically what it does is to transform that certain *NetworkObject* to a non-authoritative server side object, by giving them trust on the connected clients. With this

new approach, we can easily optimize the inputs from the player by just sending bytes corresponding to player inputs.

```
C/C++
[System.Flags]
public enum PlayerAction
{
    None = 0,
    MoveF = 1 << 0,
    MoveB = 1 << 1,
    MoveL = 1 << 2,
    MoveR = 1 << 3
}


void HandleInput()
{
    ...

    // Player movement
    PlayerAction action = PlayerAction.None;

    if (Input.GetKey(KeyCode.W)) action |= PlayerAction.MoveF;
    if (Input.GetKey(KeyCode.S)) action |= PlayerAction.MoveB;
    if (Input.GetKey(KeyCode.A)) action |= PlayerAction.MoveL;
    if (Input.GetKey(KeyCode.D)) action |= PlayerAction.MoveR;

    if (action != PlayerAction.None) Move(action, Time.deltaTime);

    ...
}
```

Notice that we are declaring the *PlayerAction* enum as a bitwise operation, by adding the shift left operation '<<'. That means that when we are trying to input, for example, *MoveF* and *MoveL,* the binary value will be 0101, which corresponds to the binary value of 1 + 4 stated in the enum declaration.

Name Tag for Players

As a player, knowing the surroundings is crucial for the decision making process inside the videogame. For this reason, it is important to add visible name tags with the player's name which allows them to easily identify the location of the other players.

To implement this functionality, Steamworks API allows access to the information inside the Steam platform. In this case, obtaining the information regarding the player name is as simple as calling a asynchronous function (a function that is being executed in the background without

blocking the rest of the code flow), *SteamClient.Name.* This function returns the user's team username, which will be saved locally inside a *NetworkVariable*, so that everyone can have the name stored and synchronized and it will be used to show it through a canvas on top of each player.

Guns, Shooting and Reloading

For this implementation, various things will be taken into account in order to implement the guns of the game.

First of all, bullets will not be physics based, meaning that they will not have bullet fall and it will not create a GameObject with physics. Instead, the gun shot will be made through a ray casting, in which the initial point will be the gun muzzle and will go towards the direction that the player is aiming.

For now, we had a simple Pistol implementation that shoots simple bullets. Now, with the idea to increase the player's arsenal, the following weapons will be implemented:

- Pistol (reworked): will shoot single bullets.
- Assault Rifle: will shoot single bullets with increased fire rate.
- Shotgun: will shoot 10 bullets at once.

So, for this purpose, a GunBase is implemented, in which each of the new weapons will inherit from it. From this base class, the Shoot() function will need to be implemented for each of the childs that inherit from this base class.

```C/C++
// Pistol Shoot() function
public override void Shoot(Vector3 origin, Vector3 direction)
{
    if (isReloading || Time.time - lastShotTime < fireRate) return;

    if (currentAmmo.Value <= 0 || Time.time - lastShotTime < fireRate)
    {
        PlayEmptyClipSFXClientRpc();
        lastShotTime = Time.time;
        return;
    }

    PlayGunShotSFXClientRpc();
    currentAmmo.Value--;

    Vector3 hitPoint = origin + direction * 999f;
```

```
    if (Physics.Raycast(origin, direction, out RaycastHit hit, 999.0f))
    {
        hitPoint = hit.point;
        if (hit.collider.TryGetComponent<IDamageable>(out var
damageable))
        {
            damageable.TakeDamage(gunDamage);
        }
    }

    if (IsServer)
    {
        SpawnTrailClientRpc(origin, hitPoint);
    }
    else
    {
        SpawnTrailServerRpc(origin, hitPoint);
    }

    lastShotTime = Time.time;
}

// Shotgun Shoot() function
public override void Shoot(Vector3 origin, Vector3 direction)
{
    if (isReloading || Time.time - lastShotTime < fireRate) return;

    if (currentAmmo.Value <= 0 || Time.time - lastShotTime < fireRate)
    {
        PlayEmptyClipSFXClientRpc();
        lastShotTime = Time.time;
        return;
    }

    PlayGunShotSFXClientRpc();
    currentAmmo.Value--;

    for (int i = 0; i < pelletCount; i++)
    {
        direction = ApplySpread(direction);

        Vector3 hitPoint = origin + direction * 999f;

        if (Physics.Raycast(origin, direction, out RaycastHit hit,
999.0f))
        {
            hitPoint = hit.point;
            if (hit.collider.TryGetComponent<IDamageable>(out var
damageable))
            {
                damageable.TakeDamage(gunDamage);
            }
        }

        if (IsServer)
```

```
        {
            SpawnTrailClientRpc(origin, hitPoint);
        }
        else
        {
            SpawnTrailServerRpc(origin, hitPoint);
        }
    }

    lastShotTime = Time.time;
}
```

Then, to show on screen the bullet trails when shooting, spawning them is required in order to show it on all client screens, that is why having a *ClientRpc* and *ServerRpc* are needed so that everyone can see the gun shots. As for the reload behavior, a *Coroutine* is started so that the reload behavior lasts until the sound effect finishes, since we are not working with animations, at least having auditory feedback when the reload sound effect finishes playing.

```
C/C++
protected IEnumerator ReloadCoroutine()
{
    isReloading = true;
    PlayReloadSFXClientRpc();

    yield return new WaitForSeconds(reloadSfx.length);

    currentAmmo.Value = maxCapacity;
    isReloading = false;
}

protected void SpawnTrail(Vector3 start, Vector3 end)
{
    GameObject trail =
(GameObject)Instantiate(Resources.Load("Prefabs/Gameplay/Items/Guns/Bull
etTrail"));
    trail.GetComponent<BulletTrail>()?.SetTrailPositions(start, end);
}

[ClientRpc]
protected void SpawnTrailClientRpc(Vector3 origin, Vector3 hitPoint)
{
    SpawnTrail(origin, hitPoint);
}

[ServerRpc]
protected void SpawnTrailServerRpc(Vector3 start, Vector3 end)
{
    SpawnTrailClientRpc(start, end);
}
```

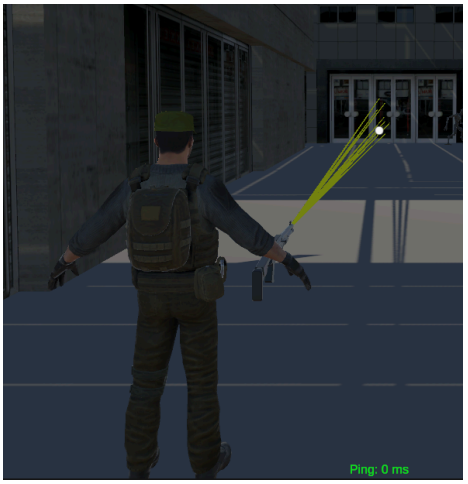


Figure 25: BulletTrails

Healing, Reviving and Medkits

Healing items are indeed important in zombie cooperative videogame. From the *PickUpItem* in the earlier iteration, the healing action was added in this iteration to give the opportunity to the player to survive longer and have a greater chance of finishing the level. Medkits will heal 80% of the amount of the health lost. The healing formula is as follows:

HealAmount = (MaxHealth - CurrentHealth) * 80 / 100

Additionally, a reviving player action has been implemented to allow the players a second opportunity to finish the level. If all the players are dead, automatically they are moved to the lobby. A reviving indicator will be displayed to show the progress needed to revive a dead player. The reviving formula is as follows:

ReviveAmount = MaxHealth * 30 / 100

Point system and Ranking

Steamworks API allows persistent leaderboards with ordered entries. These ranking tables can be used to show global rankings inside the game or the community's webpage. But since the game has not been published yet, some considerations need to be taken into account.

The game is still in production, which means that the game still does not have a Steam App ID and every single game that works with Steamworks that has not been accepted by Steam yet, needs to be working under the 480 App ID, that corresponds to the *Spacewar!* example

application. For this reason, to play a build in production it is important to have registered *Spacewar!* in a Steam account with the command mentioned in the Iteration 2. This application was created by Valve as a tool for developers to test Steamworks features before publishing the game into their website.

```
C/C++
async void RequestLeaderboard()
{
    Leaderboard? lb = await
SteamUserStats.FindOrCreateLeaderboardAsync(gameRanking,
        LeaderboardSort.Descending,
        LeaderboardDisplay.Numeric);

    if (lb != null)
    {
        leaderboard = (Leaderboard)lb;
        //Debug.Log("Leaderboard loaded: " + lb.Value.Name);
    }
}

async public void UploadScore(int score)
{
    await leaderboard.SubmitScoreAsync(score);
}

public async Task<LeaderboardEntry[]> DownloadTopScores()
{
    RequestLeaderboard();
    LeaderboardEntry[] entries = await leaderboard.GetScoresAsync(10);

    return entries;
}
```

Since many videogames were developed under the 480 App ID, when creating a leaderboard of your own, it should have a name that has not yet already been taken by another developer. If by any chance a name has already been taken, when showing the ranking it will show previously uploaded scores from a totally different videogame.

For this implementation, it has been considered the following:

- Ranking: shows the Top 10 Steam players that have finished the videogame.
- Cannot be 0 (by default, when reaching the *ExtractionZone* it gives points.
- Send the score on reaching the *ExtractionZone*.
- Should not upload if the high score is bigger than the current score. (Automatized by Steamworks)
- Certain actions give points (kill Zombies, pick up items, finishing the level…)

- Certain actions subtract points (getting hit, using medkit)
- Show on screen total score, shared by all players. (*NetworkVariable*)

Figure 26: Ranking screen

Boxes

Boxes work as obstacles in the player's walkthrough, but also include a possible way for the player to obtain advantages. These boxes can be destroyed through bullet shots and may drop a random *PickUpItem*.

These boxes are considered entities inside the game system, since they inherit from the *IDamageable* interface, which forces it to implement the functions that it has, and more importantly, to be able to take damage.

Every box can drop different items with a certain probability. This allows it to offer a different experience every time the player wants to retry a level.

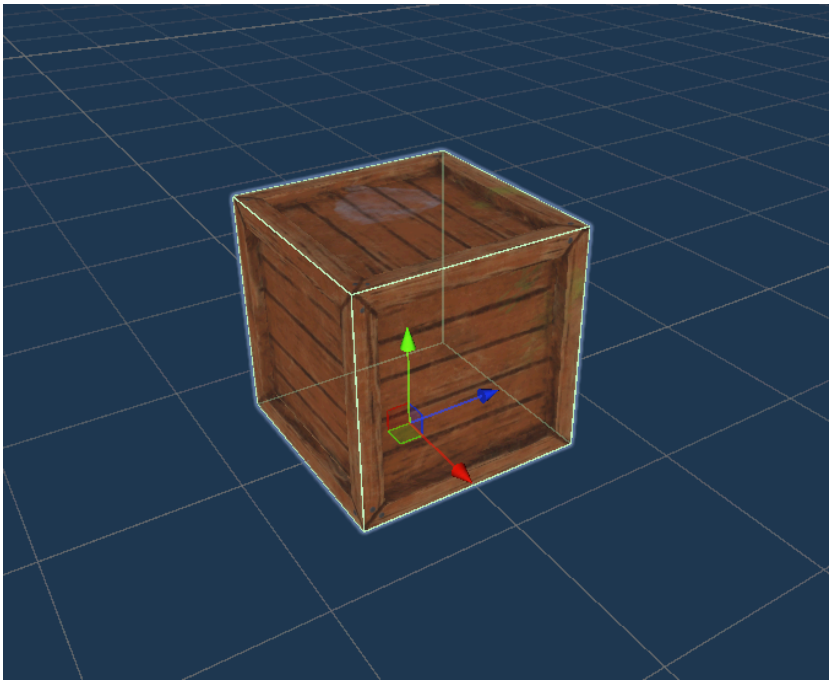


Figure 27: Box

The chance of spawning are as follows:

- Assault Rifle: 10%
- Shotgun: 10%
- Medkit: 20%
- Ammo box: 20%

Safedoor

In addition to the extraction zone, the safedoor is just an additional confirmation from the player's side to end the game. In the earlier iteration, reaching the extraction zone was enough, but now it requires the players to be in the extraction zone and the safedoor closed.

This door is indestructible. It can be opened or closed by any player nearby that is alive by simply pressing E.

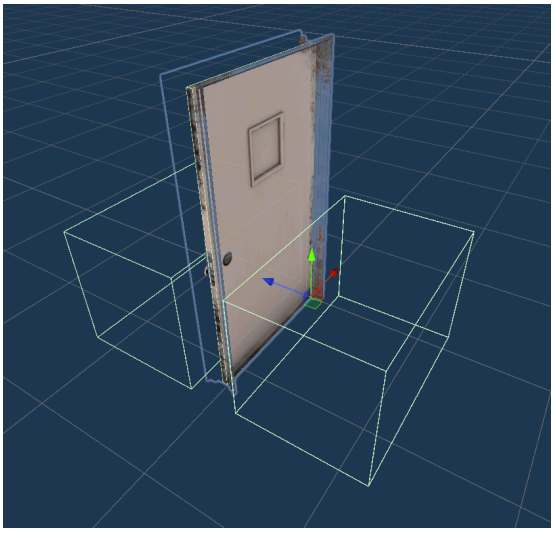


Figure 28: Safedoor

From Figure 28, notice how the colliders are positioned in front of and behind the door. When a player comes into contact with any of these colliders, they are able to interact with the door.

Hordes and Zombies improvement

Until now, there was no way to spawn the zombies dynamically, and only preloaded zombies were working. A manager that handles the spawning of hordes is needed in order to obtain a less monotonous gameplay. What it does, it basically triggers a horde of zombies which activates spawn points throughout the level if the player is nearby.

These triggers will be alarmed cars that whenever a player shoots it will then trigger a horde and many zombies will start chasing the nearest player. Alarmed cars are considered entities, which need to implement the *IDamageable* interface.

With the previous implementation of the Zombie, it needed a small rework for this iteration. For this reason, a flag to check whether it is aggressive or not has been added and changes whenever a horde is on going or when he gets shot by any player.

```C/C++
void Update()
{
    if (!IsServer) return;

    if (isHordeActive.Value)
    {
        hordeTimer += Time.deltaTime;
```

```
        if (hordeTimer >= hordeDuration)
        {
            StopHorde();
        }
    }
}


public void StartHorde()
{
    if (!IsServer) return;

    hordeSpawners.Clear();
    hordeSpawners.AddRange(FindObjectsOfType<ZombieHordeSpawner>());

    StartCoroutine(TriggerHorde());
}

IEnumerator TriggerHorde()
{
    yield return new WaitForSeconds(1.0f);

    isHordeActive.Value = true;
    hordeTimer = 0.0f;
    PlayHordeScreamClientRpc();
    ToggleAdrenalineUIClientRpc(true);
    Debug.Log("Horde started!");
}

public void StopHorde()
{
    if (!IsServer) return;

    isHordeActive.Value = false;
    hordeTimer = 0.0f;
    ToggleAdrenalineUIClientRpc(false);

    foreach (ZombieHordeSpawner spawner in hordeSpawners)
    {
        if (spawner != null)
        {
            spawner.ResetCount();
        }
    }

    Debug.Log("Horde ended.");
}
```

From the code snippet above, we can observe the core logic used to manage the start and end of a zombie horde event in a multiplayer environment using Unity's Netcode for GameObjects. The StartHorde method is responsible for initializing the event on the server side by gathering

all active *ZombieHordeSpawner* components and launching the coroutine *TriggerHorde*, which triggers the horde after a short delay.

Once the horde starts, several key actions are performed:

- The *isHordeActive NetworkVariable* is set to true, so that all clients are aware of the event.
- The horde timer is reset.
- A scream sound is played on all clients via *PlayHordeScreamClientRpc()*, providing an audio cue for the event.
- The adrenaline UI is toggled on for all players.

Similarly, the *StopHorde* method deactivates the event:

- The *isHordeActive NetworkVariable* is set to false, and the timer is reset.
- The adrenaline UI is disabled on all clients.
- Each spawner's internal count is reset for future waves.

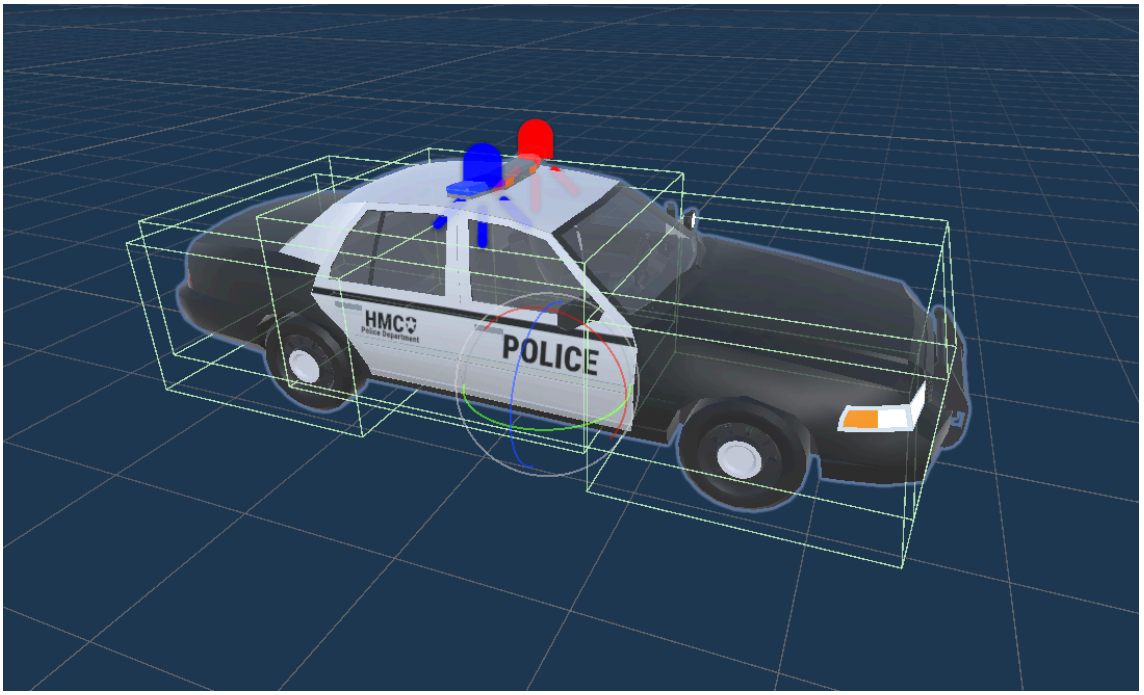


Figure 29: Alarmed car

With this new improvement, different zombie variations were implemented to this iteration. In addition to the basic zombies, fast zombies and a boss zombie were added. Regular zombies move slowly towards the player while fast zombies run quickly towards their nearest target.

The boss zombie is a giant enemy that deals significant damage if it hits a player and can withstand a large amount of damage before dying. Can only be spawned once per game.

All of the 3D assets, including the car and the zombies, were downloaded from the Unity Assets Store to speed up development and focus on gameplay implementation.

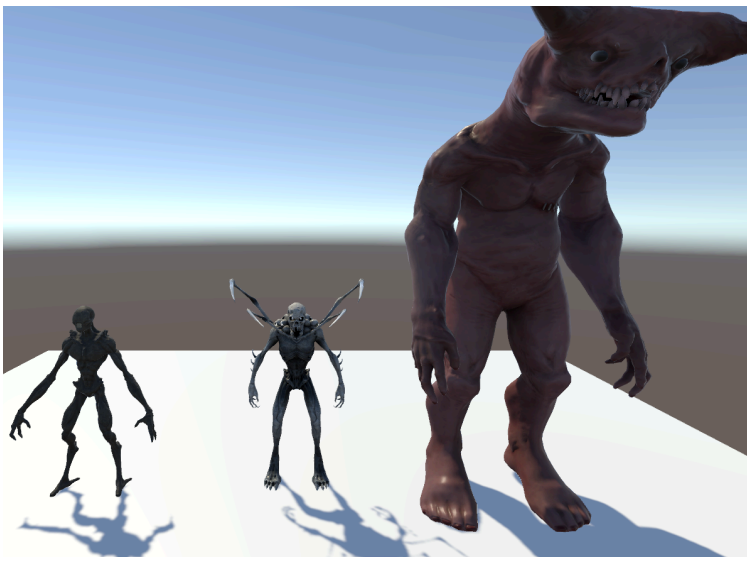


Figure 29: Zombie types. From left to right, Basic, Fast, Boss

Players alive & Disconnection callbacks

The number of players did not update correctly the amount when a player died or when a player disconnected. For this implementation, the extraction zone will check whenever a player is inside the area if all the players are alive and inside the zone. In addition to this behavior, whenever a normal client disconnects the game, the required players will be updated correctly. If by any case the client-host closes the game, presses ALT+F4, or the game crashes, it will automatically send all the players to the main menu, and get kicked from the lobby.

```
C/C++
void CheckConnection()
{
    if (!IsHost && NetworkManager.Singleton != null &&
!NetworkManager.Singleton.IsConnectedClient && !gameEnded)
    {
        gameEnded = true;
        LobbyReference.Singleton.currentLobby = null;
        SceneManager.LoadScene("1_MainMenu", LoadSceneMode.Single);
    }
}
```

Synchronization issues

Many fixes were done in this iteration since many actions were not synchronized correctly between host and normal clients.

For example, when picking up an item, from the point of view of the host the item would disappear correctly but the clients could still view the item floating in their side, sound effects not playing in client side but host could hear them, bullet trails were not spawning correctly from client side, enemies would not move on the point of view of the clients…

Basically, the synchronization problems were that the client who hosted the game could view the game perfectly where spawns and despawns are working correctly as intended but it was not synchronized from the point of view of the clients.

To address this problem, many functions were added in *ClientRpc* so that every client could see the same with a small delay. For example, this is the snippet code for the Box entity:

```
C/C++
[ClientRpc]
public void PlayBoxDestroyedClientRpc()
{
    SFXManager.Singleton.PlaySound(boxDestroySfx);
}

[ClientRpc]
void SpawnBoxEffectClientRpc()
{
    if (ps_boxDestroyed == null) return;

    GameObject box_particle = Instantiate(ps_boxDestroyed,
transform.position + Vector3.up * 0.5f, Quaternion.identity);
    Destroy(box_particle, 0.5f);
}

[ServerRpc]
public void TakeDamageServerRpc(int damage)
{
    ApplyDamage(damage);
}
```

As seen from the code snippet, all of the game details that need to be seen to all clients are done through a *ClientRpc,* and the functions that require server acknowledgement, which in this case is damaging the Box, is through a *ServerRpc*.

Iteration testing

To test the whole set of new implementations, a build has been compiled and a play testing has been run on several users. For this playtesting, the build will be played on their respective computers and see if their computer and connection is able to run under their conditions.
For those who have completed the game at least once, it will then be registered in the ranking. By gathering their feedback throughout the many versions that this iteration has undergone, several bugs have been identified and solved during the process of this iteration.

With this playtesting, feedback regarding the gameplay part of the videogame has also been collected, with overall most of them liked it and reminded them of past titles such as *Left 4 Dead 2* (which is good since most of the gameplay design is inspired by games from the genre). Despite the small visual bugs, players described the game as fun and satisfying to play, and some of them expressed interest in playing future versions of this project.

Some of the feedback regarding the network part of the game:

- The game felt really fluid with 3~4 players.
- Game was showing 0 ms for all players.

Some of the feedback regarding the gameplay part of the game:

- When killing a Zombie, its hitbox is still present.
- Car's colliders are a bit too big, making the bullets collide with "invisible" walls.
- Boxes with random drops are a bit unfair. There were game runs where no weapons were acquired.
- Maybe add recoil to the weapons.
- The instructions (read me file) did not specify that by pressing 4 the player would use a Medkit to heal. Also there is no visual feedback when reviving a player.
- Some buildings did not have their collider on, which made it possible to fall off the map.

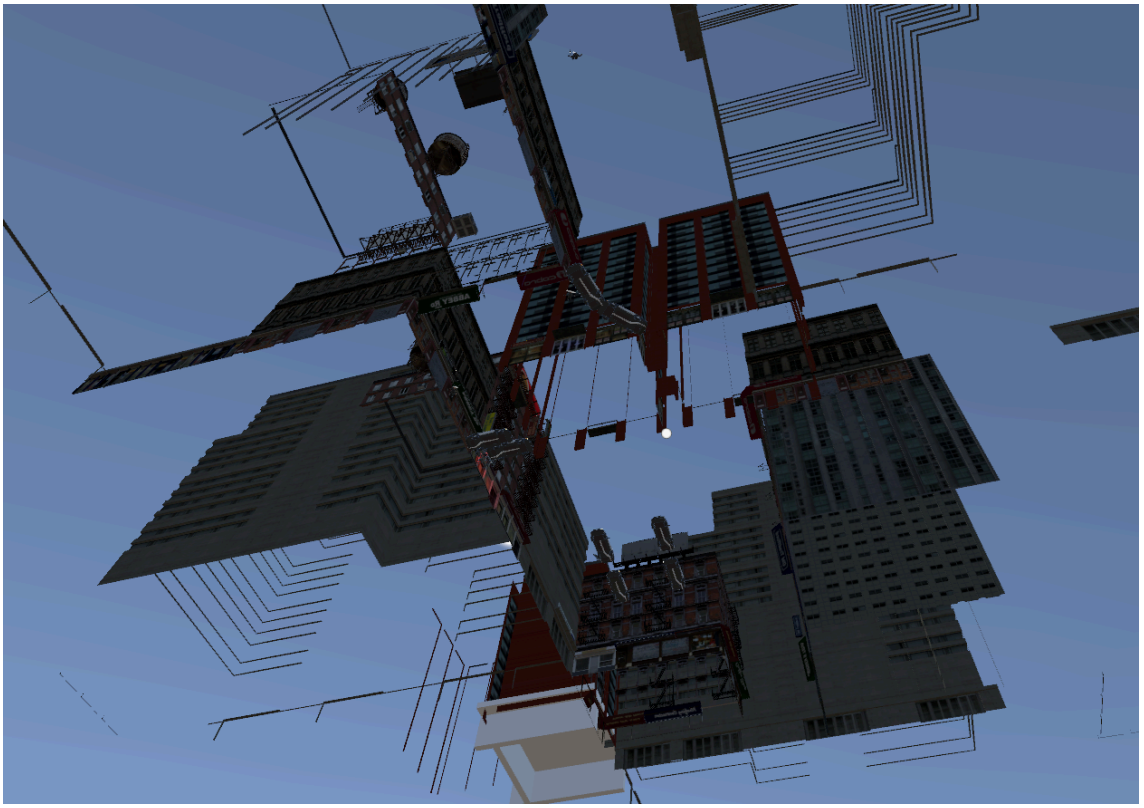


Figure 30: Out of the map bug

All of this feedback will be taken into account for the final iteration, and will be addressed and resolved the identified issues.

5.6. Iteration 5: Final testing

For this iteration, most of the things done will be fixing the identified issues found from the playtesting done in Iteration 4. And also, the integration of the cyclic level generation made by Ethan Martín Parra.

This part of the project is available in the following release as Lights Out v.0.9 and Lights Out v.1.0.

Bug Fixing and Gameplay Improvements

- All Spawner GameObjects no longer are receiving ray casts. This caused some shots done by the player to collide in “invisible” walls.
- Walls in the default level are fixed. This caused the player to fall off the map.

- Ping was also fixed. The current implementation was using the intended implementation if using Unity Transport. Since the project is using Facepunch Transport, to fix this, Ping, RTT and Packet Loss are calculated through Rpc's.
- Added particle visuals. Now when destroying Boxes, healing or reviving it spawns a ParticleSystem to give visual feedback.
- Implemented Revive Indicator. Now when attempting to revive a dead player, it displays a slider that shows the current progress needed to revive the dead player.
- Boss did not have SFXs. This is needed to properly give auditory feedback to the players.
- Car colliders are adjusted so that players can no longer shoot "invisible" walls.
- Regarding the drops from the boxes, Car can also drop *PickUpItems* to increase the chance of having a weapon in the level.
- When a Horde is triggered during an OnGoing Horde, it will increase by 1 all *ZombieHordeSpawners*.

Cyclic Level Generation

For this part of the project, the goal was to integrate the cyclic level generation system developed by Ethan Martín Parra. Since his system was originally developed in a singleplayer environment, the objective was to adapt the system to work in a multiplayer environment.

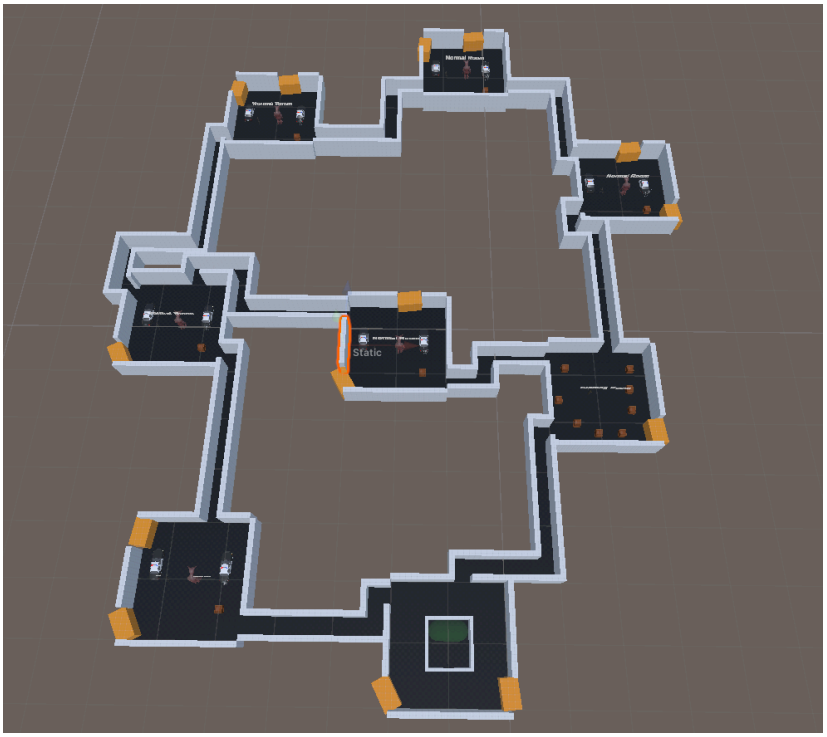


Figure 31: Cyclic Level Generation Example 1

The system generated rooms at the moment the *Scene* was loaded. Because of this, *NetworkObjects* needed to be instantiated and spawned after the rooms have been created. For this reason, empty *GameObjects* were created and saved as *Prefabs* and placed within the *Room Prefabs*. Each of these GameObjects are tagged to indicate which entity needs to be spawned on its position. Once the generation process of the Rooms is complete, then an Event is invoked to spawn the *NetworkObjects*.

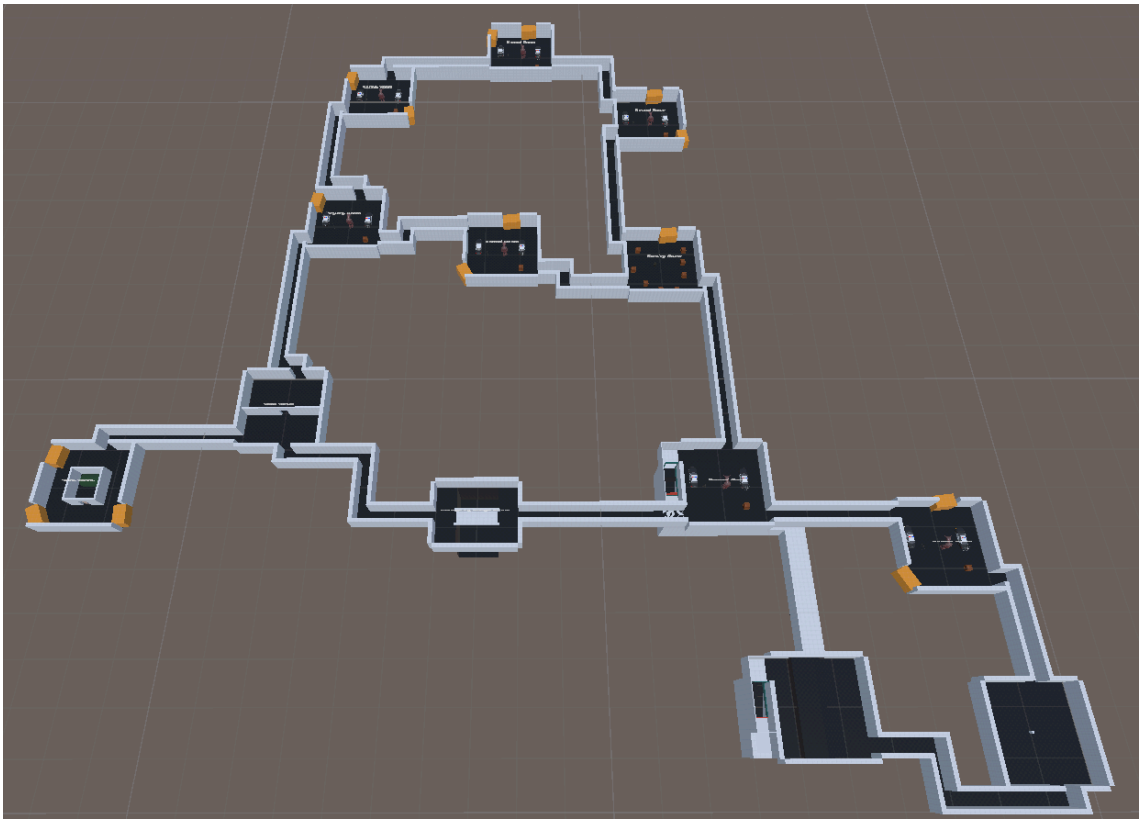


Figure 32: Cyclic Level Generation Example 2

After this setup is completed, the game is ready to be played normally as if it was playing the normal level.

Iteration testing

Like the other iterations, a playtesting session with different people was needed to gather feedback in order to gather new insights regarding gameplay, latency feeling and performance overall.

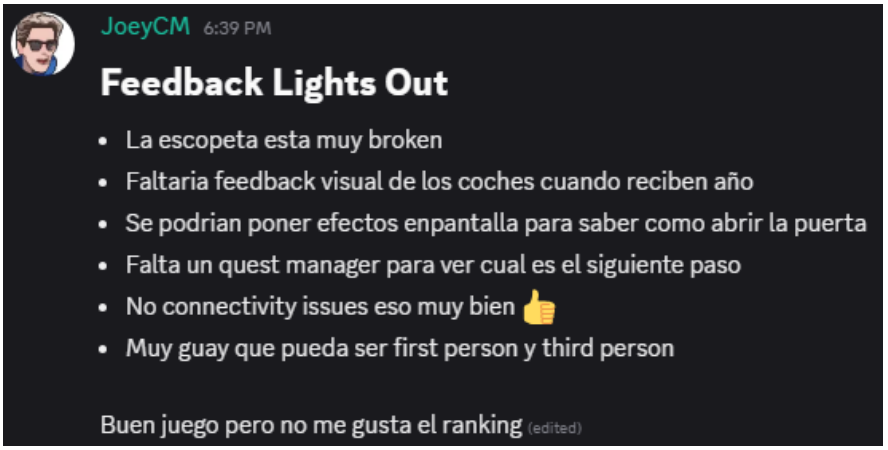


Figure 33: Final Testing Feedback 1

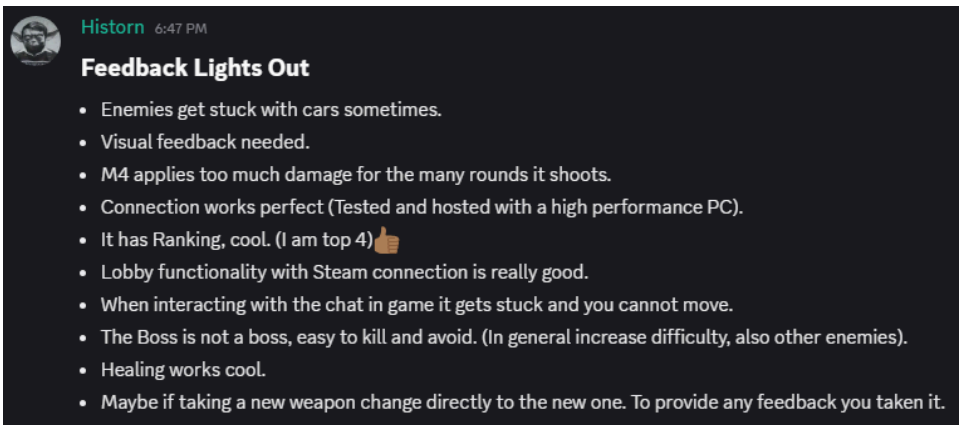


Figure 34: Final Testing Feedback 2

Overall, the average experience, from gameplay and from latency aspects, from most of the playtesters is very good, which makes this project validated.

5.7. Iteration 6: Game Details

This iteration is just to add different game details that improve the multiplayer and gameplay experience. Overall, most of them are visual things, others are essentially needed like the ping implementation. These implementations were done throughout the different iterations, which corresponds to all of the release builds published.

Master, BGM & SFX

In a videogame, regardless of whether it is a multiplayer experience or not, it should feature a way to configure the master volume, the background music volume and the sound effects volume. For this reason, a simple window has been implemented so that the user can adjust

the volume however he prefers. These adjustments will persist even when the player resets the videogame.

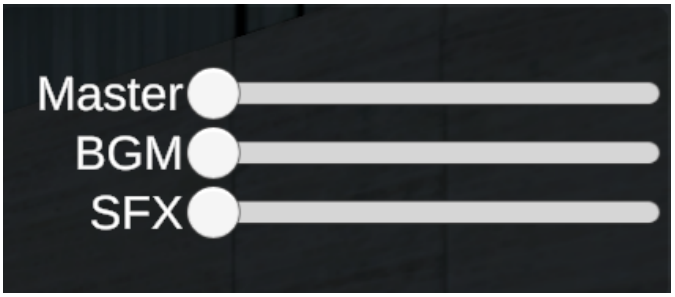


Figure 35: Sound settings

Better feedback from Zombies

To improve responsiveness and clarity of the gameplay interactions, a Particle System is used to provide a visual effect when a player hits a zombie or when the player is hit by one. In Unity, a Particle System is a component that allows to simulate effects such as smoke, fire, sparks, or in this case, blood splatter.

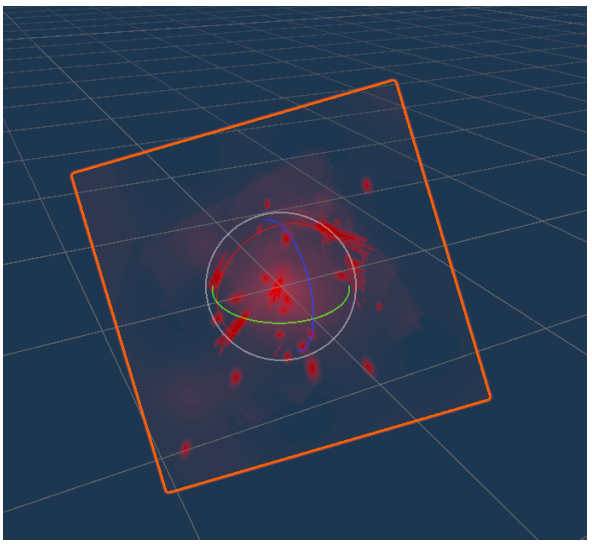


Figure 36: Blood splatter particle effect

In addition to the visual effect, an *AudioClip* is played whenever the player receives damage. In Unity, an *AudioClip* is a type of asset that stores audio information. By playing an *AudioClip*, the game provides feedback and reinforces the sense of being attacked and helps the player perceive the event if they do not immediately see the visual effect. This also helps when certain attacks are delayed because of the network connection.

There are also green particles when healing or reviving a player and brown particles with smoke when destroying a box.

In-Game User Interface

This is an important element in online games as it provides players with real-time information, helping them make quick decisions and start aware of their status during the game. In this project, the UI displays key elements such as the player's health, the number of medkits available, the current score, the elapsed time, the currently equipped weapon, the remaining ammunition, and the player's network ping.

Also, with the implementation done with the Lobby Chat done in the Iteration 2, it was recycled so that there is also an In-Game Chat that works exactly the same.

Figure 37: In-Game User Interface

Ping and Round Trip Time

Ping measures the time it takes for a signal to travel from the player's client to the game server. In online games, it is crucial to know this value which provides information about the delay that the player has in the game, which can affect the player experience.

Round Trip Time (RTT) is the total time it takes for a data packet to travel from a source to a destination and back again.

With this two definition, we can assume that the ping is:

$$Ping = RTT / 2$$

since the ping is only the time when sending the packet to the server.

The only difference is that in this project Facepunch Transport is used instead of Unity Transport. These are different low-level network transport layers used in Unity's multiplayer system and they handle the data transfer of the network.

For this reason, code like the following:

```
C/C++
rtt =
NetworkManager.Singleton.NetworkConfig.NetworkTransport.GetCurrentRtt(
NetworkManager.Singleton.NetworkConfig.NetworkTransport.ServerClientId);
```

would not work correctly, since it only works properly when using Unity Transport. The solution implemented for this project is by using *ServerRpc* and *ClientRpc* calls.

```
C/C++
[ClientRpc]
void ReturnPingClientRpc(float clientTime, int pingId, ClientRpcParams
rpcParams = default)
{
    if (pingId <= lastPingId)
    {
        receivedPings++;
        rtt = Mathf.Abs(Time.realtimeSinceStartup - clientTime) * 1000;
    }
}

[ServerRpc]
void SendPingServerRpc(float clientTime, int pingId, ulong clientId)
{
    ReturnPingClientRpc(clientTime, pingId, new ClientRpcParams
    {
        Send = new ClientRpcSendParams
        {
            TargetClientIds = new[] { clientId }
        }
    });
}
```

The ServerRpc method is called repeatedly inside an *Update()* function. This sends a request to the server, which immediately returns the call with the original client timestamp. The client

then would calculate the round-trip time (RTT) by subtracting the original time from the current real time.

To estimate the ping, the latency between client and server in one direction, the RTT is divided by 2. And the *receivedPings* variable is to control the packet loss. It does not work as a real value, but it can approximate the amount of packets lost, since it sends pings to the server every 0.1 seconds.

6. Project Validation

To ensure that the network optimization has been successfully achieved, each development iteration included a testing build focused on specific improvements. These builds were systematically tested and compared against previous versions by measuring the response time between the server and the clients. The main validation metric was the progressive reduction of response time across iterations.

Playtesting

Some of these iterations needed playtesting. The volunteered players tested each build and provided qualitative feedback regarding gameplay smoothness, latency perception, and any bugs encountered. Their input was collected through informal interviews, helping to identify areas for improvement and guiding adjustments for the next development iteration. The playtesting results can be seen on each iteration in the Iteration testing section.

Latency

Controlled latency tests were conducted using local and remote clients under varying network conditions. Ping and round-trip times were measured using in-game debugging tools and network logs. Results confirmed that average latency remained below 50 ms for local connections and under 150 ms for remote connections, meeting the expected thresholds. With the Ping implementation done in the Iteration 6, visually seeing the ping helps on the benchmark process for this section.

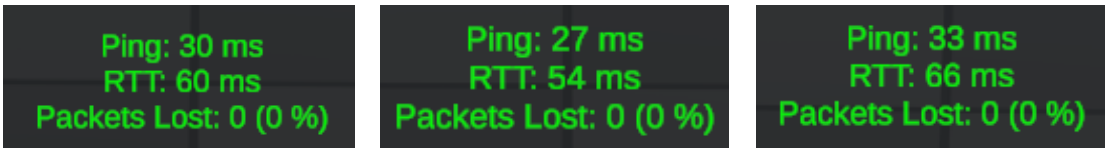


Figure 38: Average Ping, RTT and Packets Lost

As seen from Figure 38, an average of 30 ms in ping, 60 ms in rtt, and 0 (0%) of packets lost from different clients connected to a client-server.

Figure 39: Unity Profiler

In Figure 39, it is shown the use of Unity Profiler which allows to monitor in real-time the performance of the distinct components of the PC while the application is running. With this tool, we can check the amount of bytes sent per frame and computer performance.

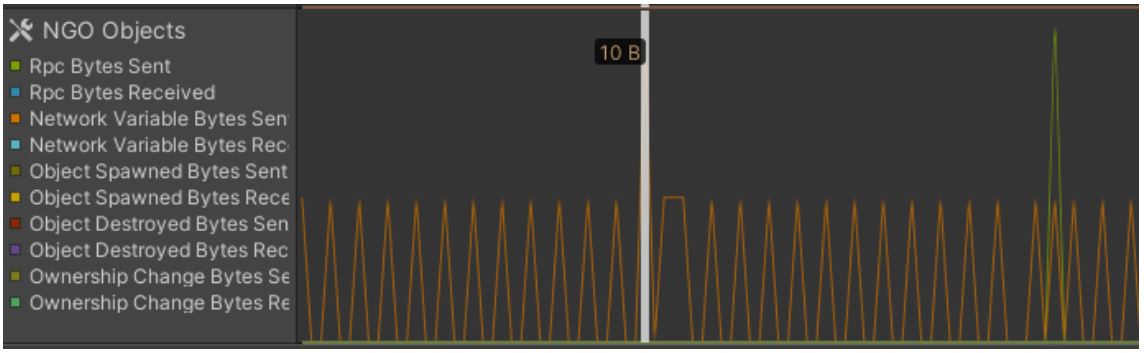


Figure 40: Maximum amount of bytes sent per frame

For this project, specifically the network data managed by Netcode for GameObject is shown in Figure 40. From the graphic, we can see that the maximum amount of bytes sent per frame is 10 bytes, which indicates an optimized synchronization and low bandwidth consumption among all the NetworkObjects in the scene.

Latency is also calculated through pings and the round trip time, in which every client will send a Rpc towards the server and receive back a confirmation message. The time elapsed between

the send and the received message is the round trip time, and if we divide it by 2, we get the ping. This can also be seen in more detail in the Ping section in Iteration 6.

Synchronization & Packet Loss

Object replication and player state synchronization were validated by running stress tests with flooding with many zombie entities in the level. Test cases included weapon switching, shooting, and movement updates under varying latency to confirm accuracy. While every spawner in the game spawned hundreds of zombies while slowly reducing the amount of zombies to see which is the recommended threshold for this type of game. Also, by despawning certain objects depending on the distance with the nearest player to free object allocations in the server helps with the synchronization process.

Scalability

Alongside the tests done with spawning several zombies, up to four simultaneous players were connected (which is the amount intended) to assess stability under typical gameplay conditions in four different computers. For this section, lag spikes and PC performance was taken into account. And as the same as the synchronization section, by spawning hundreds of zombies and reducing the amount of spawns until it finds a stabilized gameplay.

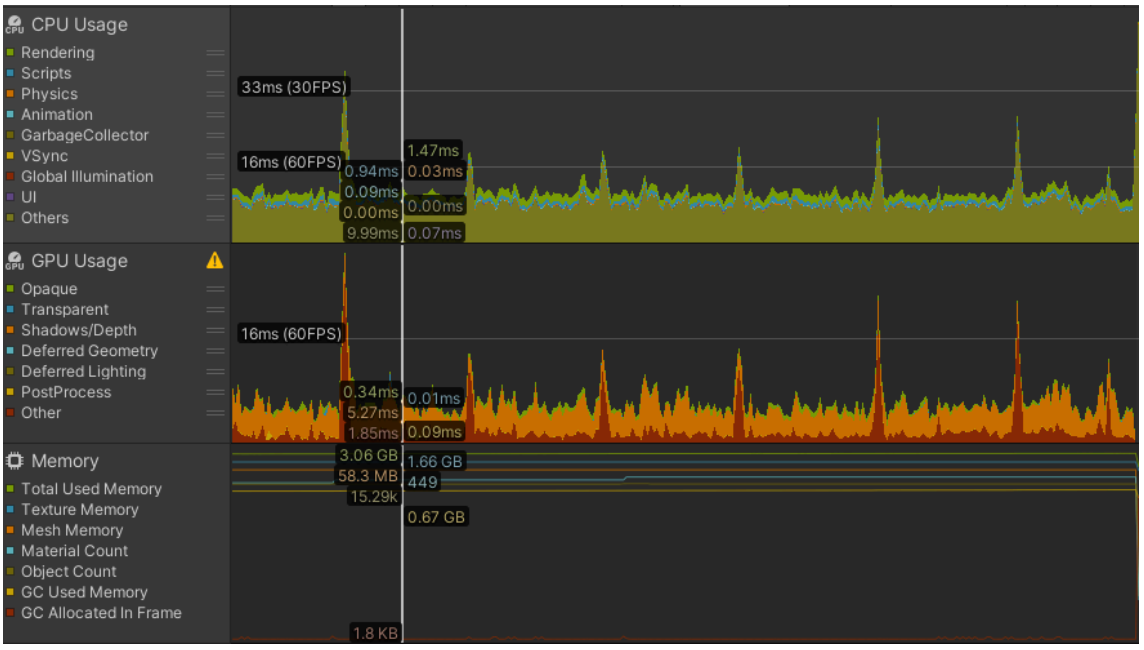


Figure 41: Computer performance

In Figure 41, we can see a graphic from Unity's Profiler of a specific frame. The graph displays CPU, GPU and memory consumption, helping to identify bottlenecks and optimize resource management. This tool also helps in terms of knowing how many entities should be on screen to have a smooth gameplay equally for all players connected. And with the current build, playing with 4 players is good enough for it.

7. Conclusions

The development of this project has provided valuable insights into the design and implementation of an online multiplayer videogame using the Steamworks API. Although the initial objective was not a success, after transitioning to Netcode for GameObjects, throughout the process, various challenges related to network synchronization, player management, and the integration of Steam services were successfully attained.

With the use of Steamworks API, the project benefits from established features such as friend invitations, and lobby management, which greatly simplify the user experience and reduce the complexity of developing a multiplayer infrastructure from scratch.

Additionally, this project served as an opportunity to deepen knowledge in client-server architecture with p2p transport (by using Facepunch transport), real-time communication, and gameplay programming. Despite certain limitations, such as the lack of dedicated visual assets and some features pending of implementation, the final result meets the initial objectives and lays a solid base for any future improvements and expansions.

Overall, the work done has contributed to understanding the practical aspects of developing an online game and highlights the advantages and challenges of integrating third-party APIs in game development rather than a custom netcode done from scratch.

The transition from a custom netcode to Netcode for GameObjects, successfully led into creating a functional prototype of a cooperative online game that features a procedural environment and real-time interaction for up to 4 players has been accomplished. This prototype offers a cooperative experience where one can invite up to 3 more friends to enjoy a zombie game, in which the levels are generated procedurally with real-time interaction when dealing with the enemies while handling dynamic objects, synchronizing player actions and maintaining game states. In the game, a variety of entities can be found, in which players interact with, such as getting damaged, killing them or destroying obstacles or even reviving dead players. In terms of game state, it detects if an OnGoing horde is currently activated, which transmits to all the players by activating part of the UI and auditory feedback so that they notice that a horde has been activated.

The average latency for all the connected clients is about 30 ms in ping, 60 ms in round-trip time and 0% packet loss, (if they are at least on the same country or nearby) which accomplishes the fact that the game can be played with up to 4 players from different places.
With the addition of a cyclic level generation, created by Ethan Martín, the multiplayer game adapted this system to further improve the replayability factor of the game.
And finally, with the integration of Steamworks API, some of its functionalities were added to the game thanks to it. Thanks to Steamworks' Lobby system, the game does not require to develop from scratch a relay server, which facilitates the communication between peers and allows the peer 2 peer connection. By also adding the chat system functionality by using Steam's channels which can be used for the Lobby and for the InGame scene. With the integration of Steam's overlay, players can also use it to invite their friends.
Also it features a ranking system which Steam saved through the API. Players are able to upload their score when finishing a run which allows them to competitively obtain the highest score possible.

In conclusion, this project not only achieves its technical objectives but also contributes valuable insights into the development of a scalable and efficient online experience. This work has a solid foundation for future development and a personal milestone in understanding real-time networking challenges in game design and development.

With the current base and knowledge gained, future iterations or even a new project can now be used to build a stable multiplayer videogame.

7.1 Future Lines

Several improvements need to be considered to enhance the quality and experience of the videogame. To begin with, having a dedicated artist that designs the visual aspects, providing better models, textures, and animations. Throughout the whole project, all the assets were retrieved from the Unity Asset Store.

Also, implementing ammunition management for each weapon remains a pending feature that would add more depth and realism to the gameplay. Additionally, the UI does not properly change the ammunition for each weapon every time they swap weapons for the clients. Currently, it only works for the client who is hosting the game.

Regarding sounds, they are currently playing in 2D. Everyone hears the same sounds regardless of distance (an exception for this is the pick up item who only plays for the player who picks up the item). For a future iteration, sounds should be played in 3D so depending on the distance from the audio source, the intensity of the sound will differ from players.

Then, when an ongoing game session is currently playing, it is also needed that players to join an ongoing game session would greatly increase accessibility and user engagement. Right now, the game once it is started, no users can rejoin the game if disconnected.

And finally, integrating a functional proximity voice chat system would enhance communication and cooperation among players and also increase the immersive aspect of the game.

8. Bibliography

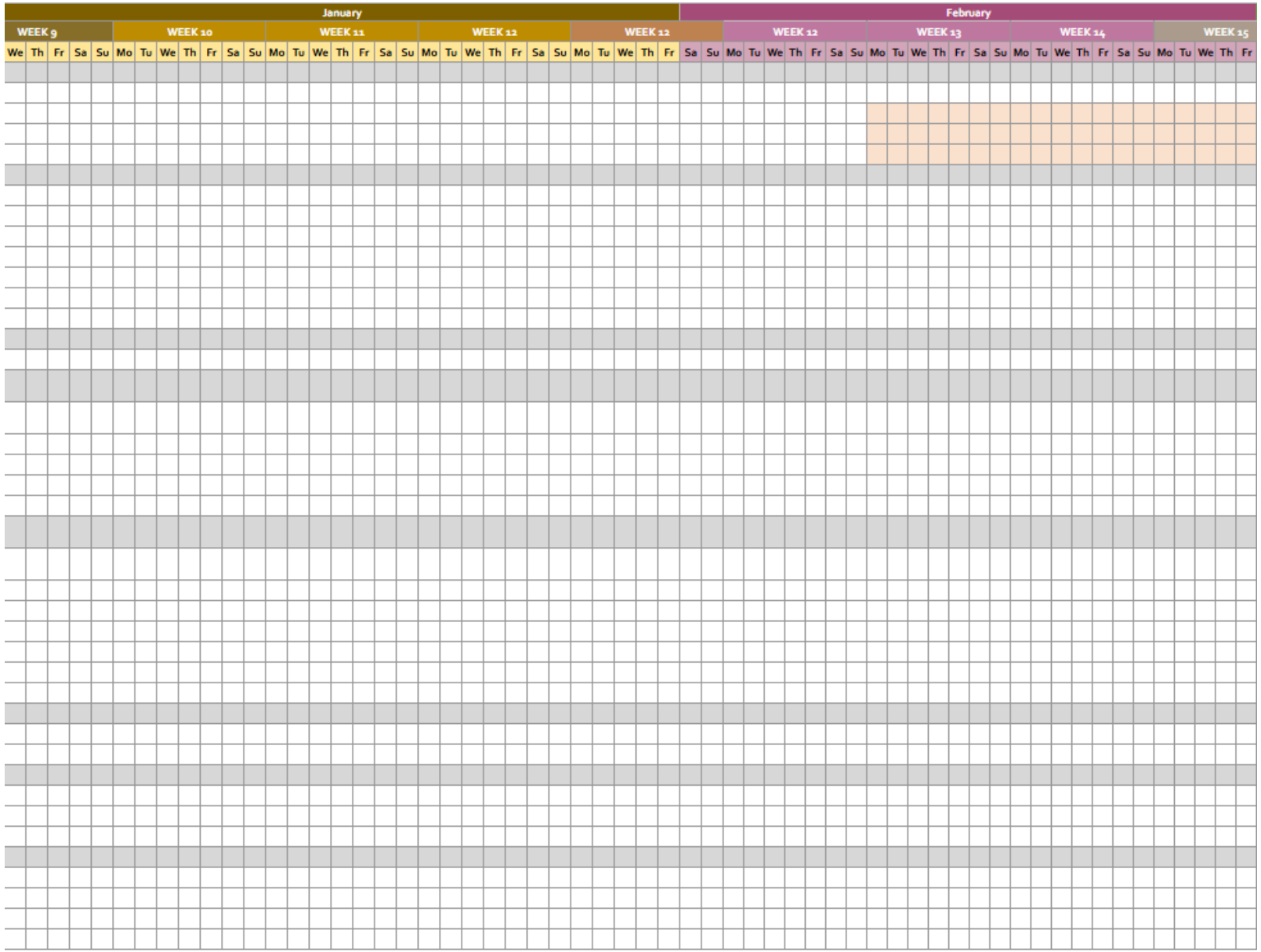
- Unity. (2025). Netcode for GameObjects. https://docs-multiplayer.unity3d.com
- Epic Games. (n.d.) Networking overview for Unreal Engine. Retrieved March 5, 2025 https://dev.epicgames.com/documentation/en-us/unreal-engine/networking-overview-for-unreal-engine
- Gambetta, G. (2024). Fast-paced multiplayer. www.gabrielgambetta.com
- Bowman, G. (2024). Gaffer on Games. www.gafferongames.com
- GameMaker Studio. (2022, June 30). Easy multiplayer tutorial. https://gamemaker.io/es/tutorials/easy-multiplayer-tutorial
- GameMaker Studio. (2024). GameMaker Language (GML) - Networking. https://manual.gamemaker.io/lts/en/GameMaker_Language/GML_Reference/Networking/Networking.htm
- Gambetta, G., & Sanjay, M. (2016). Multiplayer game programming: Architecting networked games. Addison-Wesley Professional. ISBN 978-013-403430-0
- Godot Engine. (2025). Tutorials - Networking. https://docs.godotengine.org/en/stable/tutorials/networking/index.html
- Roundy, J. (2025, February 23). Assess the environmental impact of data centers. Retrieved February 23, 2025 https://www.techtarget.com/searchdatacenter/
- Barragán, R. (2021, December 19). Back 4 Blood se actualiza: campaña para un jugador sin conexión, mercaderes y más. Meristation. https://as.com/meristation/2021/12/19/noticias/1639904344_722815.html
- Newzoo. (2025). Top 20 PC games. Retrieved March 5, 2025 https://newzoo.com/resources/rankings/top-20-pc-games
- Noticias Jurídicas. (2025, March 5). Cómo está creciendo el mercado del juego en línea en 2024. https://noticias.juridicas.com/contenido-patrocinado/articulo/19566-como-esta-creciendo-el-mercado-del-juego-en-linea-en-2024/
- Mohebatzadeh, O. (2021, May 24). TCP vs. UDP: What's the difference? https://www.linkedin.com/pulse/tcp-vs-udp-whats-difference-omid-mohebatzadeh/
- Zinn, J. (n.d.) Why video game server hosting is the key to online multiplayer games. OVHCloud. Retrieved March 4, 2025 https://us.ovhcloud.com/resources/blog/multiplayer-gaming-server-hosting/
- SilverCervy. (n.d.). So is the netcode for this game really that awful? Reddit. Retrieved March 4, 2025 https://www.reddit.com/r/remnantgame/comments/16ebj9d/so_is_the_netcode_for_this_game_really_that_awful/
- Statista. (2024, May 23). Industria de videojuegos: Ventas online en España por tipo de producto. https://es.statista.com/estadisticas/1113661/industria-de-videojuegos-ventas-online-en-espana-por-tipo-de-producto/
- Statista. (2024, February 29). Online gaming. https://www.statista.com/topics/1551/online-gaming/#topicOverview
- Valve Corporation. (2024, August 12). Source multiplayer networking. https://developer.valvesoftware.com/wiki/Source_Multiplayer_Networking
- Climatica. (2024, February 28). Videojuegos: Concienciar sobre el cambio climático. https://climatica.coop/videojuegos-concienciar-cambio-climatico/
- Generalitat de Catalunya. (2024, May 2). Factors de emissió associats a l'energia. https://canviclimatic.gencat.cat/es/actua/factors_demissio_associats_a_lenergia
- Bluetti Power. (n.d.). Cuánto consume un ordenador de sobremesa. Retrieved March 15, 2025

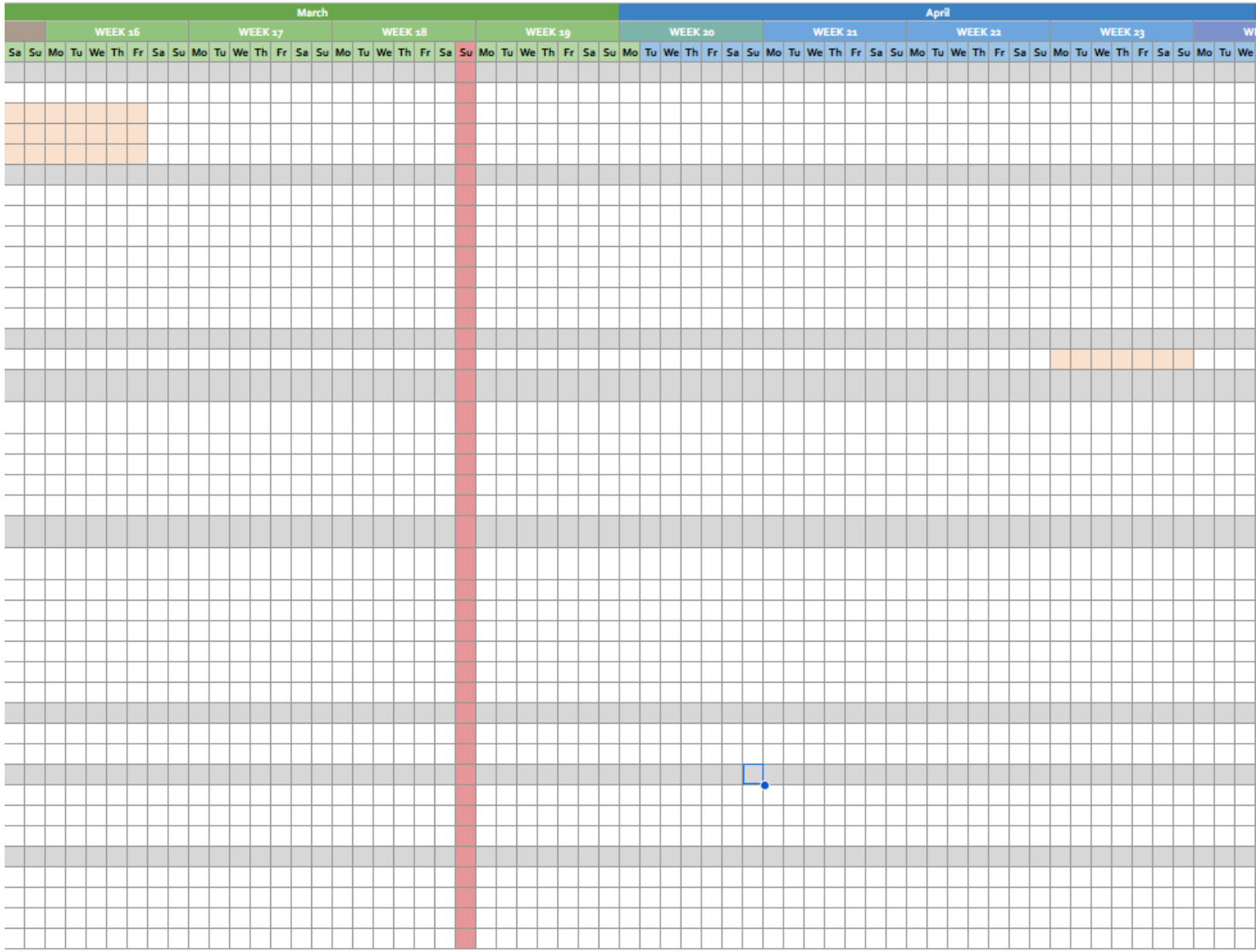
https://es.bluettipower.eu/en/blogs/base-de-conocimientos/cuanto-consume-un-ordenador-de-sobremesa
- YouTube. (2023, Nov 16). BuncyrlBu-0. https://www.youtube.com/watch?v=BuncyrlBu-0
- Square Enix. (2013, November 16). Why the netcode issue exists and why it cannot be fixed. https://forum.square-enix.com/ffxiv/threads/119219-Why-the-netcode-issue-exists-and-why-it-cannot-be-fixed
- XIV.Dev. Retrieved March 21, 2025. https://xiv.dev/
- Amazon Web Services. Amazon GameLift. Retrieved March 21, 2025 https://aws.amazon.com/gamelift/
- Tarifaluzhora.es. Retrieved March 21, 2025 https://tarifaluzhora.es/
- Bernier, Y. (2003). Latency compensating methods in client/server in-game protocol design and optimization. https://www.semanticscholar.org/paper/Latency-Compensating-Methods-in-Client-Server-and-Bernier/330071040ca858ca710a24a03915366fcd46f021

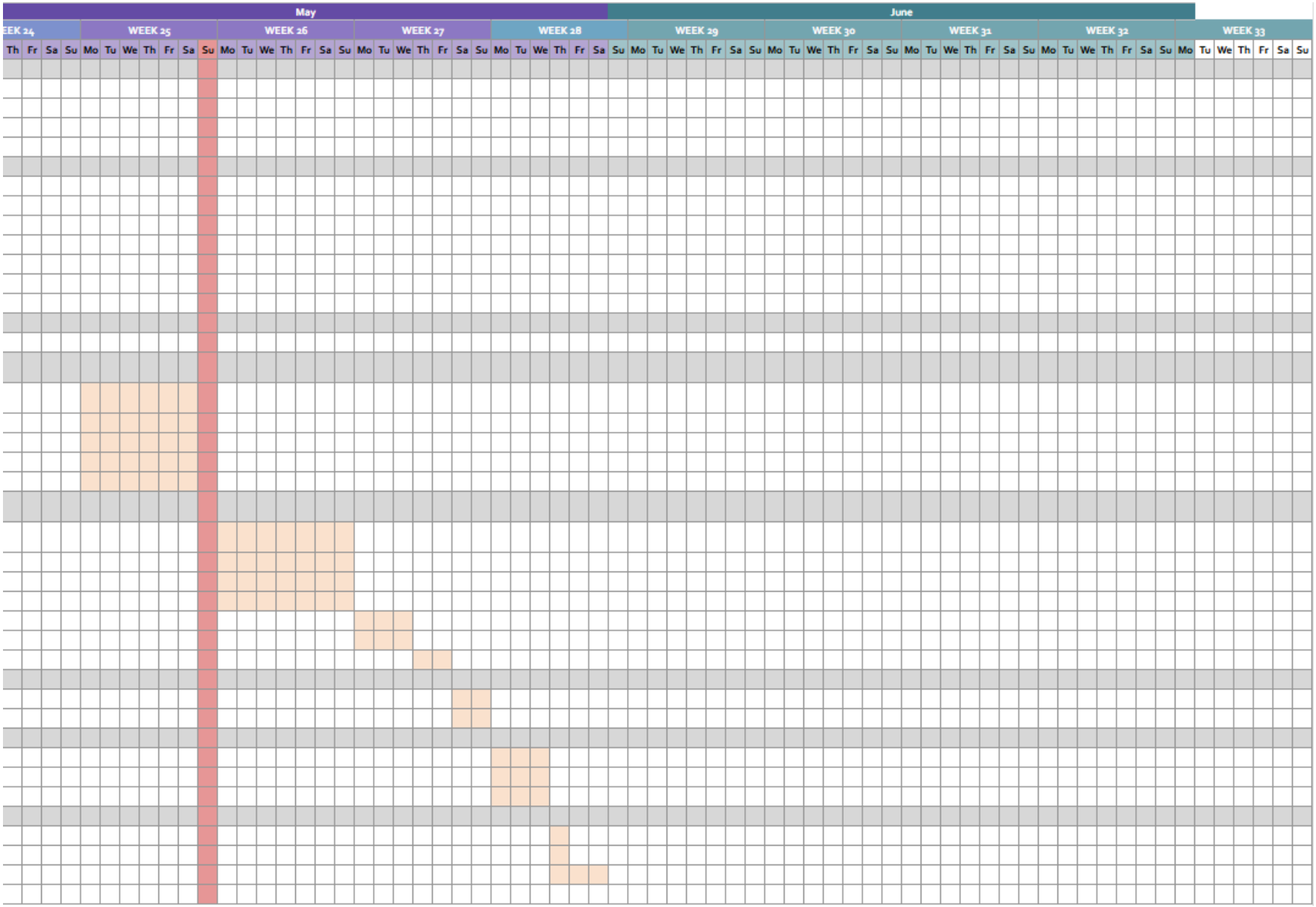
8. Annex

8.1. Gantt Chart

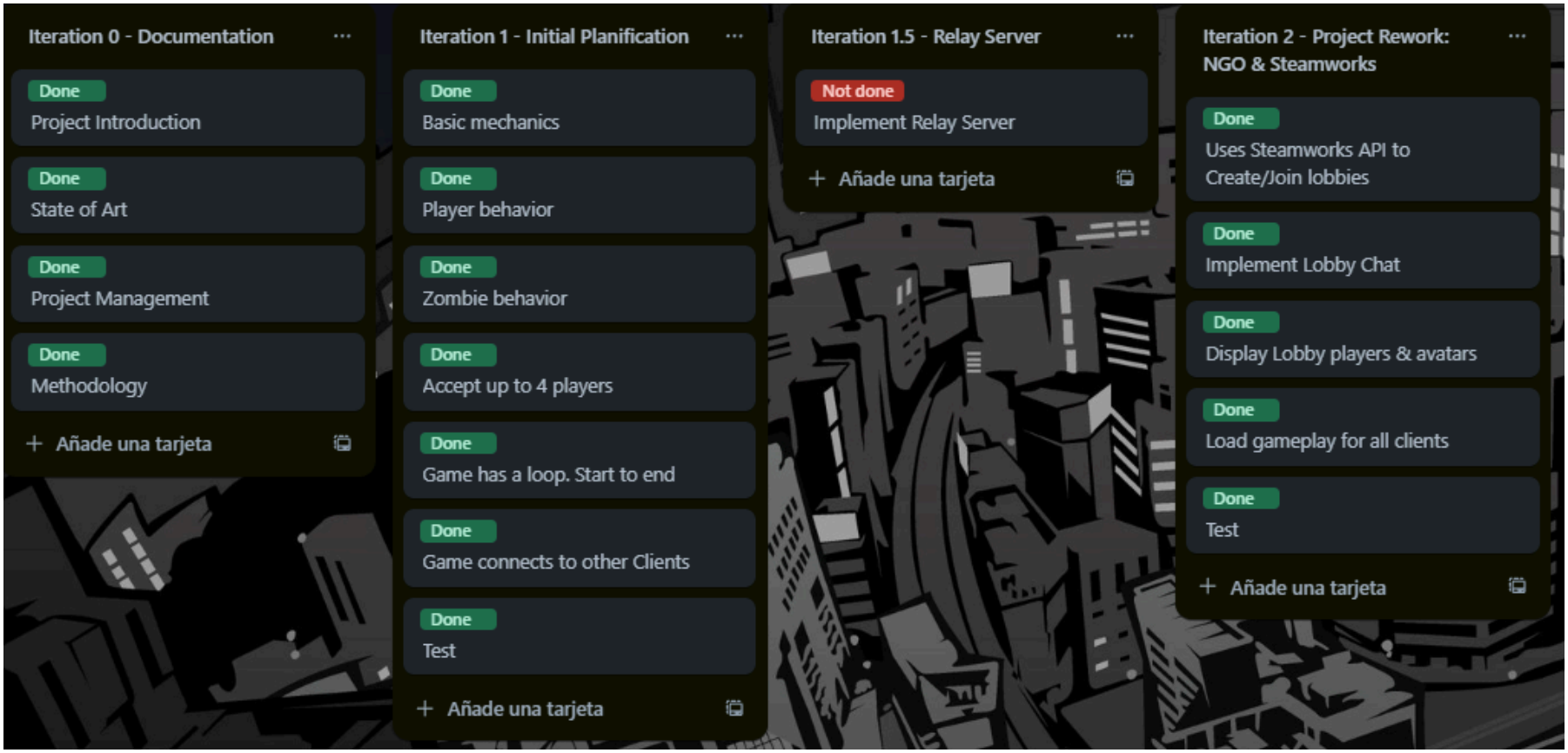
STEP	TASKS	START DATE	DUE DATE	DURATION (DAYS)
0	**Iteration 0 - Documentation**			
0.1	Introduction	11/18/2024	11/30/2024	13
0.2	State of Art	2/10/2025	3/7/2025	28
0.3	Project Management	2/10/2025	3/7/2025	28
0.4	Methodology	2/10/2025	3/7/2025	28
	Iteration 1 - The Beginning of the End			
1.1	Basic mechanics	11/4/2024	11/8/2024	5
1.2	Accept up to 4 players	11/5/2024	11/6/2024	2
1.3	Player behavior	11/4/2024	11/8/2024	
1.4	Zombie behavior	11/4/2024	11/8/2024	
1.5	Game has a loop. Start to end	11/4/2024	11/15/2024	12
1.6	Game connects to other clients	11/5/2024	11/6/2024	2
1.7	Initial UI	11/18/2024	11/19/2024	2
	Iteration 1.5 - Relay Server			
1.5.1	Implement Relay Server	4/21/2025	4/27/2025	7
	Iteration 2 - Project Rework: NGO & Steamworks			
2.1	Uses Steamworks API to Create/Join lobbies	5/5/2025	5/11/2025	7
2.2	Implement Lobby Chat	5/5/2025	5/11/2025	7
2.3	Display Lobby players & avatars	5/5/2025	5/11/2025	7
2.4	Load gameplay for all clients	5/5/2025	5/11/2025	7
2.5	Test	5/5/2025	5/11/2025	7
	Iteration 3 - Improving Game Loop functionalities			
3.1	From Sockets (C#) to Netcode for GameObjects	5/12/2025	5/18/2025	7
3.2	Adapt Player	5/12/2025	5/18/2025	7
3.3	Adapt Zombie	5/12/2025	5/18/2025	7
3.4	Implement Weapon ammunition	5/12/2025	5/18/2025	7
3.5	Implement Level generation	5/19/2025	5/21/2025	3
3.6	Breakable props	5/19/2025	5/21/2025	3
3.7	Test	5/21/2025	5/22/2025	2
	Iteration 4 - Additional implementations			
4.1	Implement Voice Chat	5/23/2025	5/25/2025	3
4.2	Test	5/23/2025	5/25/2025	3
	Iteration 5 - Final Testing			
5.1	Bug fixes	5/26/2025	5/28/2025	3
5.2	Clean code	5/26/2025	5/28/2025	3
5.3	Test	5/26/2025	5/28/2025	3
	Iteration 6 - Game Details			
6.1	Add BGM/SFX settings	5/29/2025	5/29/2025	1
6.2	Improve UI	5/29/2025	5/29/2025	1
6.3	Improve Game Mechanics	5/29/2025	5/31/2025	3
	Total			218

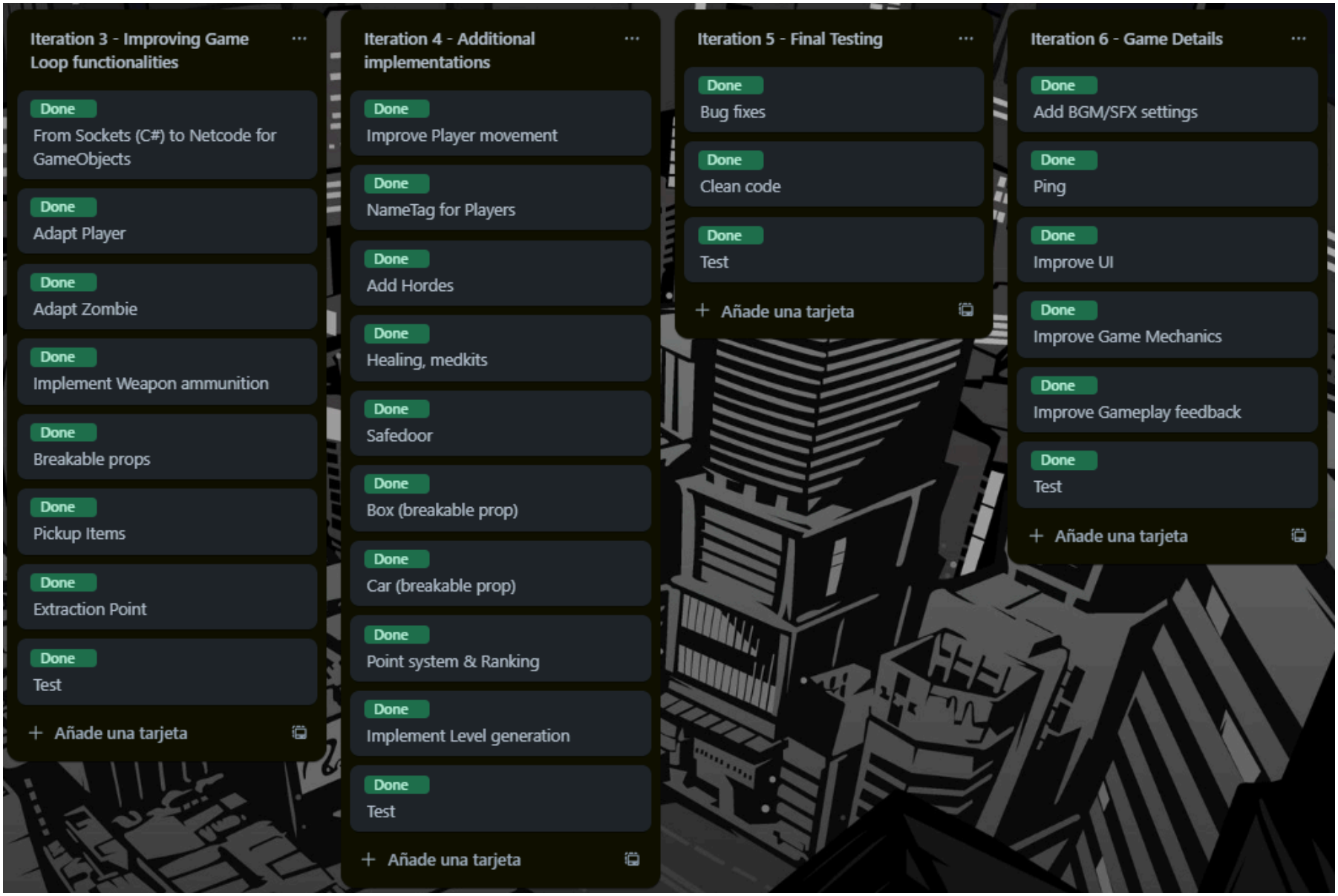

January
February
WEEK 9
WEEK 10
WEEK 11
WEEK 12
WEEK 12
WEEK 12
WEEK 13
WEEK 14
WEEK 15
We Th Fr Sa Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr


March
April
WEEK 16
WEEK 17
WEEK 18
WEEK 19
WEEK 20
WEEK 21
WEEK 22
WEEK 23
Sa Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su Mo Tu We


May
June
WEEK 24
WEEK 25
WEEK 26
WEEK 27
WEEK 28
WEEK 29
WEEK 30
WEEK 31
WEEK 32
WEEK 33
Th Fr Sa Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su

8.2. Trello


Iteration 0 - Documentation
Done
Project Introduction
Done
State of Art
Done
Project Management
Done
Methodology
Añade una tarjeta
Iteration 1 - Initial Planification
Done
Basic mechanics
Done
Player behavior
Done
Zombie behavior
Done
Accept up to 4 players
Done
Game has a loop. Start to end
Done
Game connects to other Clients
Done
Test
Añade una tarjeta
Iteration 1.5 - Relay Server
Not done
Implement Relay Server
Añade una tarjeta
Iteration 2 - Project Rework: NGO & Steamworks
Done
Uses Steamworks API to Create/Join lobbies
Done
Implement Lobby Chat
Done
Display Lobby players & avatars
Done
Load gameplay for all clients
Done
Test
Añade una tarjeta


Iteration 3 - Improving Game Loop functionalities
Done
From Sockets (C#) to Netcode for GameObjects
Done
Adapt Player
Done
Adapt Zombie
Done
Implement Weapon ammunition
Done
Breakable props
Done
Pickup Items
Done
Extraction Point
Done
Test
Añade una tarjeta
Iteration 4 - Additional implementations
Done
Improve Player movement
Done
NameTag for Players
Done
Add Hordes
Done
Healing, medkits
Done
Safedoor
Done
Box (breakable prop)
Done
Car (breakable prop)
Done
Point system & Ranking
Done
Implement Level generation
Done
Test
Añade una tarjeta
Iteration 5 - Final Testing
Done
Bug fixes
Done
Clean code
Done
Test
Añade una tarjeta
Iteration 6 - Game Details
Done
Add BGM/SFX settings
Done
Ping
Done
Improve UI
Done
Improve Game Mechanics
Done
Improve Gameplay feedback
Done
Test
Añade una tarjeta