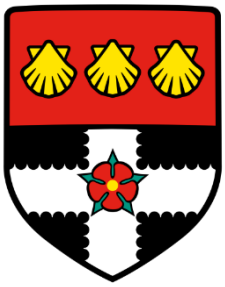

University of Reading

Department of Computer Science

Multiplayer Networked Game

Kristupas Stalmokas

Supervisor: Xiang Li

A report submitted in partial fulfilment of the requirements of

the University of Reading for the degree of

Bachelor of Science in *Computer Science*

May 15, 2025

Declaration

I, Kristupas Stalmokas, of the Department of Computer Science, University of Reading, confirm that this is my own work and figures, tables, equations, code snippets, artworks, and illustrations in this report are original and have not been taken from any other person's work, except where the works of others have been explicitly acknowledged, quoted, and referenced. I understand that if failing to do so will be considered a case of plagiarism. Plagiarism is a form of academic misconduct and will be penalised accordingly.

I give consent to a copy of my report being shared with future students as an exemplar.

I give consent for my work to be made available more widely to members of UoR and public with interest in teaching, learning and research.

Kristupas Stalmokas

May 15, 2025

Abstract

The rapid growth of the multiplayer gaming industry has sparked a significant demand for fair, robust and scalable online experiences. However, real-time, physics-based games continue to endure consequential technical challenges in delivering efficient synchronization, network resilience and security. In this project, these challenges are addressed through designing and implementing a modular multiplayer networked game within the Unity game engine, drawing on a client-server architecture with server-authoritative logic, Unity's networking framework and services. Advanced networking techniques such as client prediction, server reconciliation, reconnection functionality and dynamic parameter adjustment are employed for seamless and fair gameplay, even during suboptimal network conditions and clients with intermittent connectivity. Unity also offers a tool for network condition simulation, combined with a custom network stress test scene implementation allowing for thorough testing and evaluation of synchronization drift, latency, bandwidth usage, fairness under packet loss, reconnection success and security. Results prove that the solution achieves low synchronization drift and upholds responsive and fair gameplay across a variety of simulated latencies and player counts, with predictable scaling in bandwidth usage and reconnection success rate stayed reliable. Security tests confirmed the validation of the server-authoritative logic and authentication due to correct blocking of unauthorized actions, spoofing attempts and maintained data integrity. The findings enunciate the effectiveness of this combination of modular architecture along with resilient networking strategies as the outcome is a fair and resilient multiplayer experience for two or more players.

Keywords: Multiplayer Networking, Unity, Real-Time Synchronization, Client Prediction, Server Reconciliation

Report's total word count: 18913

Table of Contents

Chapter 1: Introduction

1.1 Background

Multiplayer games play a major part in the modern gaming industry and have revolutionized how games are played and interacted with. From the simple cooperative experience that Pong provided in the 1970's arcades to the local couch co-op that Halo offered in 2001. Currently there's a wide range of complex, globally connected online multiplayer games like Minecraft, League of Legends and Fortnite that are easily accessible on computers, consoles with some even available on mobile phones (Tikhomirov A 2023).

Multiplayer gaming has evolved from its simple roots as a niche hobby and has set standards for social interaction, engagement and, with a market valued in the billions, commercial success. This is due to the implementation of multiplayer driving retention of players and revenue in a rapidly growing industry. Successful multiplayer games are built on advanced online networking functionality that requires technical prowess for reliability and scalability. The aspect of multiplayer, be it the primary focus or an inserted mode of play, allows multiple people to connect online and partake in a gaming experience together. Multiplayer also extends the gameplay with the inclusion of social interaction, cooperation and competition (Tikhomirov A 2023).

The game industry is a major sector in the entertainment industry with significant growth. In the four years between 2017 and 2021 (see Figure 1) the annual revenue has increased from $120.4 billion to $214.2 billion and has been predicted to rise to $321.1 billion by 2026 (PwC 2022). In comparison with the movie industry's revenue, which reached $99.7 billion in 2021 without TV revenue, and $328.2 billion with it (Motion Picture Association 2021). Video games, with multiplayer games being a cornerstone sector, are rivalling even cinema and may catch up in the near future. This places the video game industry as an alluring path for programmers and associated specialists to venture their careers into.

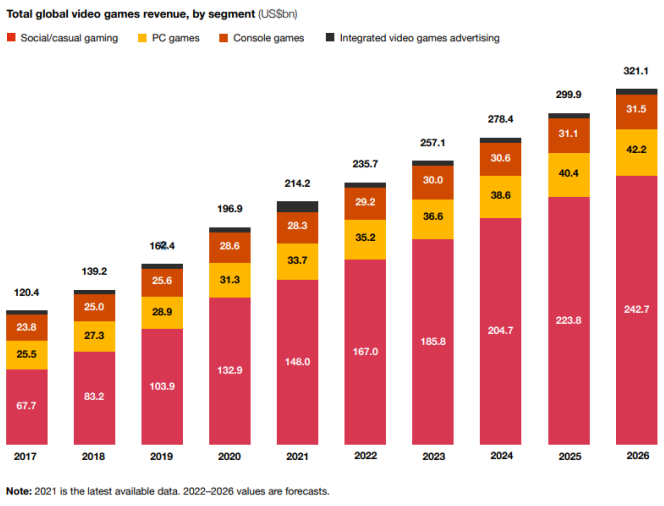

Figure 1: Global Game Revenue in USD (PwC 2022).

Development of these multiplayer games, especially real-time and physics-based games, is faced with unique technical challenges not present in single-player or even local multiplayer games. The most important of which being making sure every player experiences the same, consistent game that is kept fair for all players. Issues of network latency, bandwidth limitations and potential cheaters are vital to address to deliver this seamless experience. Games that feature physics-based or real-time gameplay have these issues even more pronounced as seemingly minor discrepancies in the synchronization of player's states will result in different experience between players (Tikhomirov A 2023).

Different genres of multiplayer games require fundamentally different network architecture for efficient operation. Turn-based games such as chess or card games can function with high delay as the integrity of updates are more important than speed of transmitting them. In contrast, fast-paced real-time shooters like Valve's Counter Strike demand much lower delays and higher update frequencies (Valve Corporation 2024). Physics-based gameplay which Rocket League employs has the network architecture built around ensuring complex collisions and interactions are consistent and synchronized between players. Simple 2D games have more modest network requirements than 3D games but still need careful consideration – the network architecture must be specifically designed for the gameplay mechanics and expectations of responsiveness (Tikhomirov A 2023).

Past programmers had to develop games from the ground up but as games evolved so did creation tools. There is a plethora of game engines to speed up the process of development, for instance Unity, Source, Godot and Unreal. Common technical issues involved in online multiplayer games may also be handled somewhat by these engines, such as Unity's Netcode for GameObjects (NGO**)** framework for multiplayer connections (Tikhomirov A 2023; Unity Technologies 2025b).

Furthermore, the multiplayer gaming field is constantly seeking to advance networked game development regarding the technical and commercial demands of multiplayer gaming. Another advancement is respecting the social, ethical and legal considerations. This is done by ensuring player data is protected with suitable privacy regulations, ensuring fair play amongst players via anti-cheat mechanisms. Another direction is sparking a positive online community through management of potential risks of cheating, data breaches and network disruptions (Tikhomirov A 2023).

1.2 Problem Statement & Project Scope

Despite multiplayer games boasting massive popularity and commercial success, the substantial technical challenge of ensuring a seamless, synchronized and fair experience for all players still stands. This problem not only negatively impact the player experience and fairness, but can also deteriorate industry standards and the

reputation of developers. For easier comprehension, this problem can be separated into four distinct, yet closely interconnected issues:

- Synchronization: ensuring the game states of every player remains consistent, particularly regarding physics.
- Network issues: minimizing delays that spawn from latency, bandwidth limitations and unstable connections which can all disrupt the experience.
- Security: preventing cheating and protecting player data – both vital for fair play and trust between players.
- Scalability: supporting larger player numbers from without any reduction in performance or reliability of the network.

These issues are deeply connected - security vulnerabilities can be caused by poor synchronization while mismanagement of network latency can desynchronize game states. Moreover, these issues introduce social, ethical, and legal risks such as unfair gameplay, breaches of privacy and a toxic community which are vital to address. One way to address these is through risk assessment and effective strategies to mitigate these risks, thereby delivering a robust solution that ensures an enjoyable, secure and fair multiplayer gaming experience.

The goal is a robust multiplayer game where every player experiences a synchronized, enjoyable and fair online game world, without any interference from the network conditions is the goal. Latency should be minimized and cheating be prevented while supporting reliable scalability for small groups of players. This solution must also communicate data usage policies and maintain a safe and welcoming online community by enforcing a code of conduct to the users. All third-party libraries, frameworks and assets will be used in compliance with their licences and the responsibility for user-generated content must be clearly defined to govern legal liability.

To precisely target the aims and focus of the solution strategically and be able to critically evaluate the most prominent challenges faced in networking and synchronization, a thorough project scope is defined. This scope also builds a solid foundation for future expansion.

Core features and functionality included in the project's scope:

- Multiplayer networking via a suitable network architecture using a game engine and capable data synchronization and communication between players.
- Lobby/matchmaking system in which players can create, search, join and leave lobbies with support for player ready status, player list and host/client roles.
- Game state synchronization of player positions, physics states, game scores and events carried out efficiently.
- Gameplay featuring unique and complex interactions between players.

- Network resilience to common network issues of latency and packet loss, including functionality to reconnect disconnected players and temporary game state persistence.
- Security and fair play by basic anti-cheat measures alongside the architecture for crucial game logic and secure management of player data.
- Stress testing through a dedicated network stress test mode for evaluating latency effects, usage of bandwidth and overall robustness of the network under heavier load.
- Managerial architecture for script management kept modular to ensure scalability and maintainability.

1.3 Aims & Objectives

The aim of this project is to design and implement a real-time, physics-based multiplayer game with support for multiple players to connect, interact and compete online whilst ensuring synchronization is kept seamless with fair play and resilience to network issues while providing a secure, efficient and scalable online experience with regards to the ethical, legal and social considerations.

To achieve this aim, the specific and measurable objectives below will require addressing:

1. Allow players to find each other online and agree to start the game.
    - Implement a system where players can find, create, join and leave online lobbies before starting the game.
    - Include ready functionality in which all players must be ready and agree to begin a game.
    - Display relevant information in the UI, such as host/client roles, player ping and ready values and lobby data (lobby name, player count).

2. Efficiently share the game state while minimizing the network traffic and the potential for cheating.
    - Ensure every player's view of the game world (including player character positions, physic states and game scores) are synchronized in real-time through network variables, RPCs and custom data serialization.
    - Optimize the transmission of data to reduce the bandwidth usage of the network by only sending data that is necessary (data that has changed since last state) or efficient data serialization.
    - Implement architecture specific logic for vital events in-game like player movements, physics and variable handling (game score) to prevent cheating.

3. Ensure the system is resilient when dealing with unreliable or intermittent network connections.
    - Enable packet loss and latency handling so the game remains playable despite suboptimal network conditions.

- Implement reconnection functionality to allow players that disconnected either manually or due to poor connection to rejoin so that the game can restore the last state of the game when they were connected and continue the experience.
- Provide temporary game state storage and restoration so the reconnection process is seamless.

4. Ensure real-time synchronization between player's views of the game world.
    - Use logic based on the chosen network architecture and regular reconciliation of the game state to maintain every client's state consistency.
    - Incorporate lag compensation, interpolation/extrapolation network techniques to keep the impact of network delays on the gameplay experience minimal.
    - Carry out regular testing and validation on synchronization accuracy under various network conditions.

5. Perform evaluation on the system's performance and efficiency.
    - Design and conduct stress tests on the network to gauge the effects of latency, bandwidth usage and system efficiency under heavier than regular load.
    - Analyse results of the stress tests to validate whether the solution meets the objectives for synchronization, fairness and resilience.

6. Sustain the standards established for the legal, ethical and social challenges.
    - Ensure the system remains compliant with regulations concerning data protection and third-party asset licensing.
    - Broadcast data usage policies and impose a code of conduct to avoid toxicity in the online community.

This list of objectives is directly mapped to the core challenges of the project and will be validated via implementation, testing and analysis.

1.4 Solution Approach

This section outlines the solution approach to reach the problem stated previously, the project's aims its objectives. By combining established networking models in the multiplayer game industry along with efficient synchronization techniques and modular software structure, a resilient real-time multiplayer game with physics in Unity that delivers scalability and fairness will be produced. This approach demonstrates innovation in dynamic network adaptation, modular architecture and networking communication methods.

The challenges of multiplayer games and the further difficulties of real-time, physics-based gameplay are addressed via a modular, object-oriented architecture in Unity. This is combined with a client-server network topology utilizing Unity's NGO framework. This approach establishes server-authoritative design with secure connectivity and matchmaking with integrated player authentication, a lobby system and relay servers.

Real-time synchronization is overcome as the system makes use of network variables, custom serialization techniques and remote procedure calls (RPCs) to achieve efficient data transmission and security. The network is made resilient through advanced networking techniques, for example client prediction and reconciliation with dynamic parameter adjustment based on connection intermittence as well as a reconnection system with state recovery. Fairness, performance and security with respect to ethical and legal considerations around player conduct and data privacy are also thought out in this solution.

1.5 Summary of Contributions & Achievements

This project's outcomes have been successfully designing and implementing a real-time, physics based multiplayer networked game in Unity with support for 2 or more players. The system features an efficient client-server architecture, modular and OOP based manager design accompanied by a seamless integration of Unity's Authentication, Lobby and Relay services.

Development of this project has achieved an advanced lobby and matchmaking system, real-time state synchronization utilizing RPCs and Unity's NetworkVariables, server-authoritative logic for vital game events. Additionally, dynamic networking techniques such as client-side prediction, reconciliation and reconnection with state recovery and a dedicated network stress test mode that evaluates bandwidth usage the performance of the network, bandwidth consumption and in-game effects of latency.

Extensive testing yields a clear demonstration of the system's stable performance under various network conditions. Synchronization, fairness and playability has been maintained even during latency and packet loss due to the dynamic network parameter adjustment for clients with high latency, aided by secure data handling and anti-cheat measures.

These contributions ensure a seamless, fair and enjoyable online experience found in the solution. Overall, this project advances best practices in multiplayer game development offers practical insight for building scalable, fair and reliable online experiences just as meeting all the criteria and pursuing novel features to make headway in multiplayer networking.

1.6 Organization of the Report

The remainder of this report is sectioned into seven chapters, each addressing a key element of the project:

- Chapter 2: Literature Review
    - Reviews the state-of-the-art in multiplayer networking and the techniques found within: real-times synchronization, physics-based game networking and development alongside critical analysis of existing solutions with their relevance to this project.

- Chapter 3: Methodology
    - Provides a deep insight into the design and implementation approach, including system architecture, algorithms, networking strategies, anti-cheat mechanisms and ethical considerations.

- Chapter 4: Results
    - Presents the outcomes of the implementation and testing carried out on system performance, synchronization accuracy and network resilience under differential conditions.

- Chapter 5: Discussion & Analysis
    - Interprets and evaluates the outcomes with discussing the significance of the results and comparing them with existing work. Limitations of the current solution are addressed here.

- Chapter 6: Conclusions & Future Work
    - Summarizes and reflects the main findings and achievements of the project, coupled with directions for research and development in the future.

- Chapter 7: Reflection
    - Offers a personal reflection on the learning experience, skills developed and challenges encountered throughout the project.

Supplemental material in the form of code Listings and extra Figures are included in the appendices.

Chapter 2: Literature Review

2.1 Networking Fundamentals

Building a solid foundational understanding of computer networking is essential for the design and implementation of any multiplayer game. The underlying concepts of network protocols, architectural models

and standards for communication directly affect a network's performance, reliability and multiplayer game fairness. This section explores these vital networking concepts with focus towards their implications for multiplayer game development.

A computer network is a link between two or more computing devices (hosts or end systems) for the sharing of resources files or for communication. These end systems are connected via communication links and packet switches. When data is being sent from one end system to another, the data is broken up into packets and follows a path/route of communication links and packet switches to get to the receiving end system, where the data is then put back together as it was originally sent. (Kurose & Ross 2017)

One major example of a computer network is the public internet which interconnections billions of devices across the world (Kurose & Ross 2017) with around 63% of the world's population being online in 2023 (Ritchie et al 2023), demonstrating the vast scale and upmost importance of networking. The internet is the largest public network and forms the backbone of online multiplayer gaming (Kurose & Ross 2017).

2.1.1 Protocols

Activity in a network is determined by network protocols, and these are used all throughout the internet and multiplayer games. Protocols directly control how data is sent and received within a network, as well as the format and order of data that is exchanged between two or more devices that are communicating (Kurose & Ross 2017).

It is crucial that systems are created on agreed upon protocols and Internet Standards, defining protocols like the two most used protocols are the Transmission Control Protocol (TCP) and the Internet Protocol (IP) – collectively known as TCP/IP (Kurose & Ross 2017). In real-time multiplayer gaming, protocols must be considered carefully as they allow players to connect and play together smoothly. As every protocol has different rules affecting the transmission of data, they can make the networked game feel smooth and responsive or delayed and disjointed (Massive Realm 2024). The Open Service Interconnection (OSI) model show in Figure 2 outlines the various layers of networking protocols.

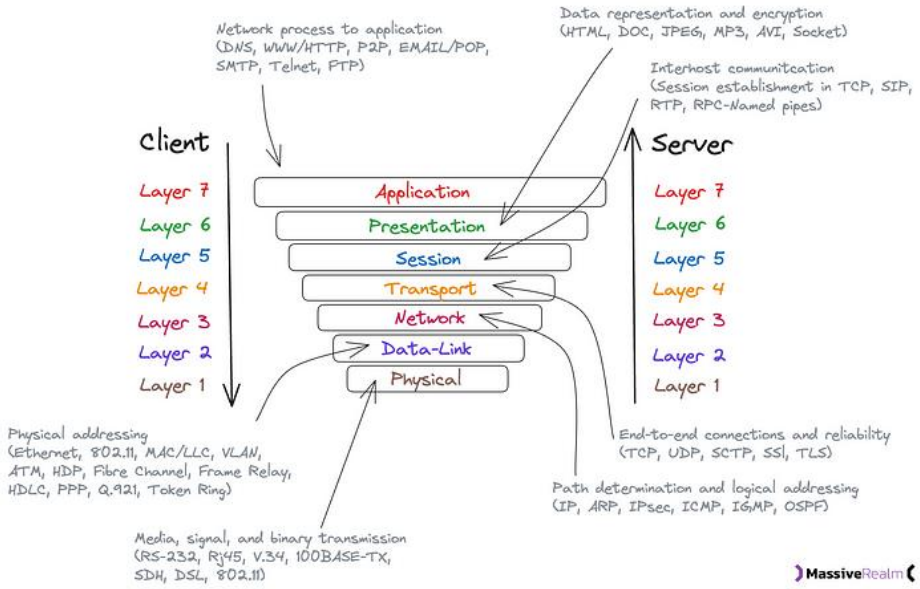

Figure 2: OSI Model for Common Network Protocols (Massive Realm 2024).

The OSI model displays the transfer of data between the different layers. A simple understanding of each layer aids in understanding the process of networks which allows multiplayer games to function smoothly. There are seven layers each with a specific role. Protocols must be carefully considered within multiplayer games as they affect how the data moves, which impacts how the game functions (Kurose & Ross 2017; Massive Realm 2024).

The choice of protocols to use in real-time multiplayer games can ensure players can play smoothly together as the protocols depict how data (such as the game state) is sent between clients. The transport and network layers are the most significant as they handle how the data packets are transmitted and how reliably. (Kurose & Ross 2017; Massive Realm, 2024).

2.2.2 Architecture

As Laiba Siddiqui (Siddiqui L 2024) puts it: 'A network architecture comprises a layered structure' where each layer breaks down tasks involved in communication into smaller sections, allowing for each layer to be responsible for a certain function in a more manageable network infrastructure. This specific structure and logical layout allow for the requirements of the network to be focused on and met most efficiently; therefore, decision-making is important and depends on the needs of the network.

Both LANs and wide area networks (WANs) require a network architecture for organization and leads to a clear view on how every element within the network interacts with each other to provide the network's requirements (Siddiqui L 2024; Geeksforgeeks 2025).

Getting into specifics, network topologies are critical for proper functionality and communication of networks and their specific requirements. Each topology provides varied benefits and drawbacks through their

implementation and protocols used. The design of a network must implement the most appropriate topology so that the system ensures efficiency, reliability and security specific for their use case (Geeksforgeeks 2025).

For a small-scale and easy to set up network, peer-to-peer (P2P) is ideal, as shown in Figure 3. Applications for file sharing, messaging and gaming use this architecture due to the cheap cost and that every client/peer can transmit data and resources with each other directly – with control for who can access that data but this does not mean this architecture is secure (Siddiqui L 2024).

The other choice is a client-server architecture, visualized in Figure 4, where clients are different from the servers and must request the server to allow them to access resources and transmit data. The server processes every request and, if approved, carries them out. Applications for data storage, security and resource management use this architecture as it provides enhanced security features (Siddiqui L 2024).

A client-server architecture is very popular for large-scale multiplayer games for the benefit of anti-cheat – control is authoritative so only the server can submit authentic updates to e.g., player movements whilst in P2P architecture peers can send any data that may be malicious or give an advantage to them in the game (Siddiqui L 2024).

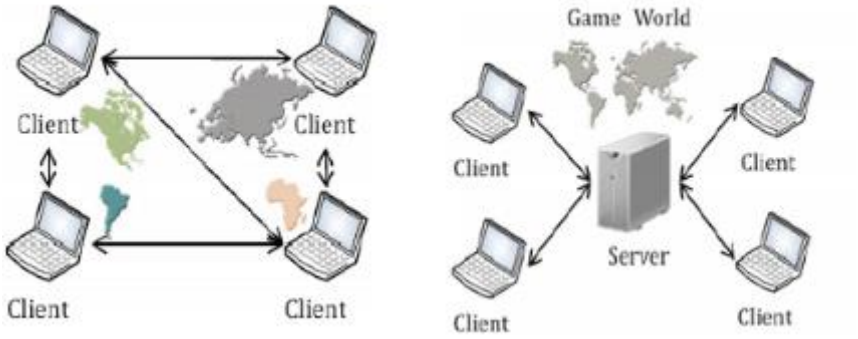

Figure 3: (left) Multi-client Peer-to-peer Connection (Xu Y et al 2014); Figure 4: (right) Multi-client Client-server Connection (Xu Y et al 2014).

2.2 Multiplayer Game Networking Concepts & Architecture

2.2.1 Protocols & Architecture in Multiplayer Games

A multiplayer game is where players share the same experience with each other. This experience appears and feels like a game in which all players are playing the same game, however in reality players are playing alternate games with their own states. For players to play the exact same game is expensive so games instead synchronize the information required to give the illusion of playing the same game as the other players (Stagner A R 2013).

A multiplayer game needs to simulate the changes in an experience because of time or player input. A singleplayer game also utilizes states in time and input from one player – a multiplayer game is the same but more complex as there is interaction between players (Stagner A R 2013).

One option for protocols in multiplayer gaming is the Transmission Control Protocol (TCP). It makes sure that every data packet reaches its destination in sequence and accurately. This protocol works in the transport layer and is great for reliable data transmission (Kurose & Ross 2017; Massive Realm 2024; Geeksforgeeks 2024).

In games where accurate data delivery in a certain order, like turn-based games or games where updates in the environment or in-game purchases, this protocol is suitable. It is possible for delays as the protocol takes time to ensure accurate and in sequence data arrival and this leads to lag in real-time games (Kurose & Ross 2017; Massive Realm 2024; Geeksforgeeks 2024).

An alternative protocol is User Datagram Protocol (UDP). This is a quick process that does not require a reply like TCP does and is used in real-time applications as speed and efficiency carry more weight than data accuracy. This protocol also functions in the transport layer and whilst it is quicker and allows for more flexibility compared to TCP. It does not ensure accurate order, delivery or avoid duplication of data packets and is suitable where some loss of data can be handled, like the positions of players in a real-time multiplayer game (Kurose & Ross 2017; Massive Realm 2024; Geeksforgeeks 2024).

While UDP is usually favoured for real-time gameplay due to its low latency, it sacrifices reliability and order. Critical data like player deaths, match results, chat or lobby management (Valve Corporation, 2015) need high accuracy which UDP does not provide, so many games implement custom layers for data reliability on top of the UDP layer, known as reliable UDP (rUDP) (Stagner A R 2013).

On the other hand, TCP's reliability has higher latency compared to UDP, potentially introducing latency spikes which must be avoided for fast-paced gameplay but can be manageable for critical data or features which do not need to be carried out as quick as possible. Regarding security, UDP is susceptible to spoofing attempts, but TCP is vulnerable to denial-of-service (DOS) attacks (Kurose & Ross 2017; Massive Realm 2024).

Unity's NGO is built on UDP for its transport layer. Low latency transmission of data that changes quickly, such as player positions, velocity and physics states can be achieved through this method. For important data, namely scoring or game state updates, Unity instead implements reliable and ordered messaging on top of UDP to maintain consistency across all clients (Unity Technologies 2024b).

TCP and UDP are traditionally relied upon in multiplayer games, but other protocols like QUIC and WebRTC are being incorporated. Primarily for browser-based and cross-platform networked games , these protocols have encryption built in, along with superior Network Address Translation (NAT) traversal. These will not be considered for this project as they are not quite mainstream due to lack of engine support, complexity and platform compatibility issues (Massive Realm, 2024). An overall comparison between protocols can be seen in Appendix Figure A.1.

For most multiplayer games, a client-server architecture is utilized – clients connect to a single server which processes the game state and acts as a middleman for messages between the clients. A client is one instance of the game running on a device and the server manages the game state, player inputs and data synchronization. This architecture provides scalability, security and control which makes it suitable for larger games with more players and data that needs to be synchronized as well as for stronger security against cheating and unauthorized access (summertimeWintertime 2022, Stagner A R 2013; Gambetta G 2024; Siddiqui L 2024).

Another architecture that can be implemented is a peer-to-peer (P2P) network – each pair of clients acts as the client and server. Each client can share their resources and data with other clients whilst also controlling which clients access their data. This architecture provides some scalability and resilience but is not able to protect clients from security breaches, either unauthorized access or in-game cheating. This makes it more suitable for smaller scale games where cheating is not a risk (such as for co-operative games where all players are on the same team – players are usually friends of the host) (summertimeWintertime 2022; Gambetta G 2024; Siddiqui L 2024).

There are many methods for players to connect with each other which depend on the architecture being used. For Local Area Network (LAN) connections, peer-to-peer is very suitable. This method is much easier and cheaper to set up and players can simply connect to each other as each player is their own host. The downsides that come from a P2P architecture such as the difficulty to prevent cheating, desync issues with lower performing internet clients and the fact that every client's IP address is visible as the clients will all be friends and under the same network (physically under one roof, small distance for sharing data). Therefore, this option very is suitable for LAN. Another downside is that that there is no rollback support for clients (summertimeWintertime 2022; Siddiqui L 2024).

An alternative to P2P is a player host where one client acts as the host and every other client connects to their server directly. The same downsides apply but the clients can only see the host's IP whilst the host can see every client's IP. The host can cheat whilst clients are less able, but one advantage is the fact that only the host's internet performance affects the server's connection reliability – one client could cause desync for the rest in the P2P network but here this is not the case. There is no upkeep like P2P so overall this is very suitable for small LAN games where players will not cheat as they all know each other (summertimeWintertime 2022).

A more advanced form of the player host setup is called a relay – a dedicated server essentially relays data from the players to the host (see Figure 5). This architecture does require some minor upkeep as there is a simple server involved for the relay of data. With this price comes a few benefits; no clients can see other clients' IP's, basic anti-cheat against the host via a hybrid model, no rollback for the host and more suitable for Wide Area Network (WAN) connections (summertimeWintertime 2022; Unity Technologies 2025a).

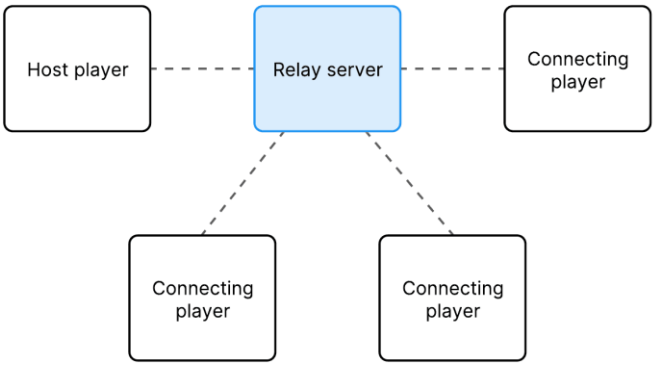

Figure 5: Relay Server in a Client-Server Model (Unity Technologies 2025a).

The most popular choice of multiplayer connection for WAN connections is via a client-server setup where, as stated previously, a dedicated server acts as the host rather than a player. This comes with more upkeep than the relay approach as the server is required for more than just relaying data between host and client – the server is the host. No players can see the IPs of other clients and cheating is much more securely prevented (summertimeWintertime 2022; Stagner A R 2013; Unity Technologies 2025a).

This architecture is also the most reliable in terms of connection and data transmission and there is a potential rollback for all clients. Despite relay severs providing better security and NAT traversal, the setup and maintenance can be costly depending on the player count and the latency of clients will be slighter increased – this would not be suitable for fast-paced competitive games (summertimeWintertime 2022; Stagner A R 2013; Unity Technologies 2025a).

Table 1: Comparative Table on Architecture.

Architecture	**Scalability**	**Security**	**Latency**	**Cheating Risk**	**Typical Use Case**
Client-server	High	High	Medium	Low	Large online games
Peer-to-peer	Low	Low	Low	High	Small LAN, Local Co-op games
Player Host	Medium	Medium	Medium	Medium	Small online games
Relay	Medium	High	Medium	Low	Any scale, Cross-NAT, privacy
Dedicated Server	High	High	Low	Low	eSports and MMO games

2.2.2 State Synchronization & Management

Netcode is the mechanism behind the synchronization of game state and events which must be kept synchronized between all devices whilst dealing with the limitations of the network. For players to play games together, all devices must be kept in sync so all players can accurately view the state of the game consistently whilst allowing for their inputs to affect the game state which needs to be synchronized to other players as well (Gao Y 2018).

This game state is the current state of the shared game and includes player positions, movements of objects, variable values and game events, such as a player moving their character or spawning collisions act on the game state, exemplified in Figure 6 (Gao Y 2018).

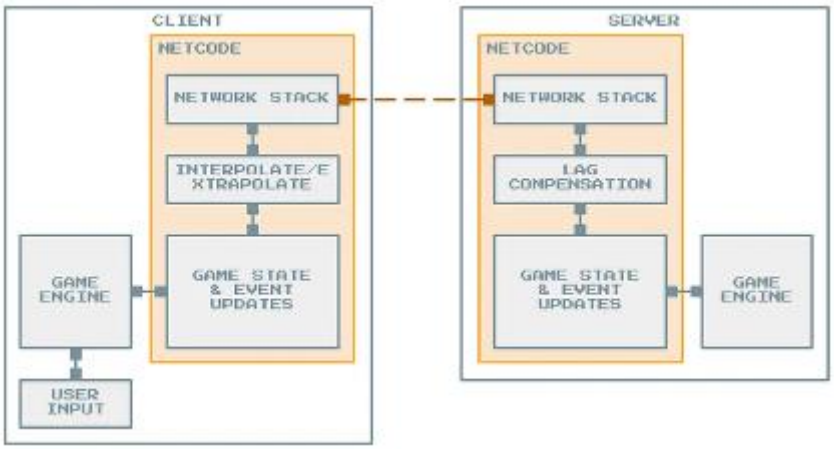

Figure 6: Netcode in a Client-Server Topology (Gao Y 2018).

The most important and complex challenge in multiplayer networking is to ensure that every client's state is in sync with the server's state. The logic and processes to make a game function will need to be handled by the server and then that data must be shared with players so that the server and clients are experiencing the game at the same state (Lim Q W 2017).

One simple way of state synchronization is to allow the clients to send updates to the server at a fixed interval - a small amount of time (30ms). This update would send the client's input. When the server receives all the inputs from every client, it can move onto the next tick and perform calculations based on those received inputs. Figure 7 provides a visualization of this update, known as the Lockstep State update (Lim Q W 2017).

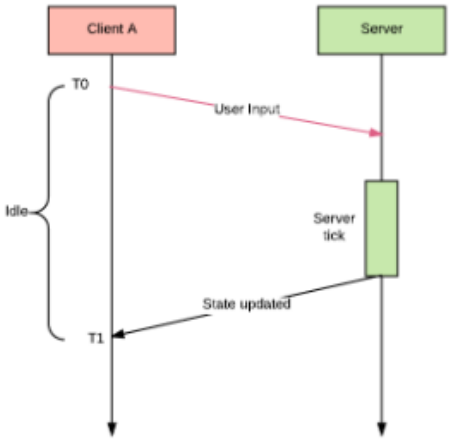

Figure 7: Lockstep State Update in a Client-Server Topology (Lim Q W 2017).

There is, however, one issue; the client will remain idle whilst it waits for the server to update the state and return it. This latency can range from 50ms to 500ms, which is a big problem for most games as any delay above 100ms is noticeable. Another problem with this method is encountered with multiple clients (Lim Q W 2017).

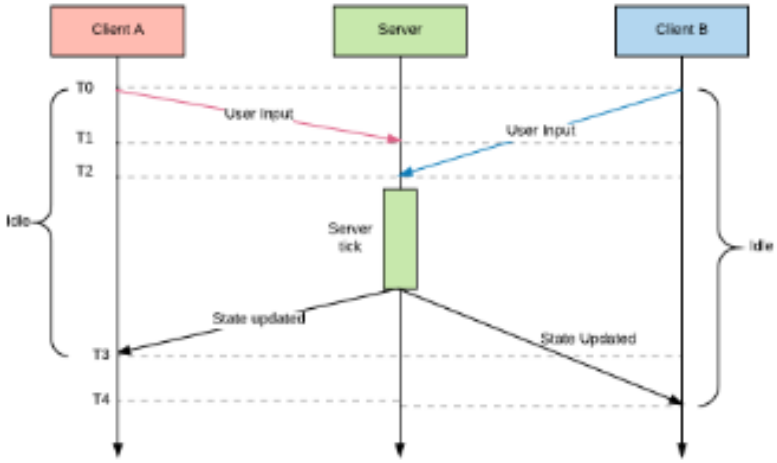

Figure 8: Lockstep State Update in a Client-Server Topology with Two Clients (Lim Q W 2017).

In Figure 8, client B has a slower connection so despite both client A and B sending input to the server at the same time, the input reaches the server after client A's input reaches the server. The server must wait for all inputs, so in this case it must wait for client B's input before performing the state update. The latency of the slowest connection player is now the delay that the game has – all players are delayed even though only one may be lagging. This is not a major issue for slower paced games, such as turn-based games and slow-paced games where a minor delay is acceptable, but for any fast-paced games this method is not suitable whatsoever considering the issues. These issues can be countered with advanced networking techniques (Lim Q W 2017).

2.2.3 Network Challenges & Techniques

Latency is the time taken for a signal to arrive at its destination from its source. To assume a signal can arrive instantly is bad judgement as this will never happen – there will almost always be some latency between devices. If this delay of signals is long enough, one device may already have moved on and be ahead of the other by the time the signal is received, causing a difference in the game state when comparing the two devices which leads to the devices no longer being synchronized (MY.GAMES 2023).

The latency of a network connection is mainly determined by route distance, packet routing delays and the protocol being used as they may require multiple trips through the route to complete data transmission (MY.GAMES 2023).

Ping is used when users provide inputs in which the signal travels to the server and then is sent back to the player with their updated game state based on their input provided. The ping rate is measured by the round-trip time (RTT) or data packets between the device and server. A higher ping means a higher delay in inputs and game state updates. RTT is a technique used to measure the time a data packet takes to travel the route from sender to receiver and then return to the sender – see Figure 9 for a portrayal. This metric is vital in multiplayer games as it affects each player's latency (MY.GAMES 2023).

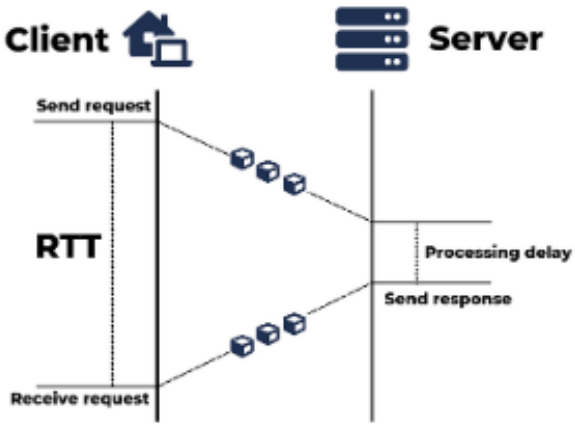

Figure 9: Round-trip Time in a Client-Server Topology (MY.GAMES 2023).

Bandwidth is the maximum volume of data that can be transferred through a network at a given time. Multiple devices under the same internet connection all transfer data and therefore more data, which typically comes with more devices, leads to a higher volume of data that is being transferred. If the network's bandwidth is smaller than this volume, then not all the data will be able to be transferred as there is not enough room. Bandwidth does not dictate the speed of the network (MY.GAMES 2023).

Lag is the delay between the inputs of a player and the reactions of the server and occurs when the network is congested with too much traffic. A player's network may slow down because of too much activity on the servers (possibly caused by insufficient bandwidth). Lag can also be caused by; high latency, a high ping rate, internet speed issues, unsuitable connection type and issues with wireless connections. Even minor delays can

negatively affect multiplayer gaming experiences, especially in real-time games like first-person shooters (MY.GAMES 2023).

Jitter is a difference of the delay time between data packets that successively arrive. Data packets arriving at irregular intervals signify network instability. A high jitter causes issues in multiplayer games as delay consistency is vital for advanced techniques that combat delay are only effective with consistent delays. High jitter negatively affects the gaming experience as it can lead to lag, movement stuttering and even packet loss which negatively impact the overall experience in fairness and responsiveness. Players with different levels of jitter will also not experience fair play as the lower jitter player has the advantage of faster registering actions (MY.GAMES 2023).

Packet loss occurs when a data packet or multiple packets do not arrive at their destination due to various reasons like traffic overload, network issues or problems with equipment/infrastructure. Necessary information can be lost leading to an interruption of gameplay; player characters can freeze; objects could disappear and other inconsistencies with the game states amongst players (MY.GAMES 2023).

Tick rate refers to the game's frequency of managing and producing data each second. In a single tick, the server performs calculations and processes which updates the game state and then sends this new state to clients. This repeats for every tick - no processing takes place between ticks (MY.GAMES 2023).

A faster tick rate means clients receive new data quicker, reducing delay and increasing responsiveness of the game. It is important to consider physics within the game as a server tick rate of say 60Hz may not be able to carry out all the physics calculations in time for each tick, causing issues with the physics simulation (MY.GAMES 2023).

Every entity in a multiplayer game must either be predicted or interpolated by clients on their local device to reduce the delay of the Lockstep state update method. There are factors to consider when choosing which technique to go for, such as game genre, physics use, player count and intended gameplay of each entity. Using a fixed update rate, the client sends input to the server and then emulates the state of the game during the delay. The technique of client prediction, showcased in Figure 10, allows the client to render the experience without having to first wait for the updated state from the server which arrives later, improving user-responsiveness significantly (Lim Q W 2017).

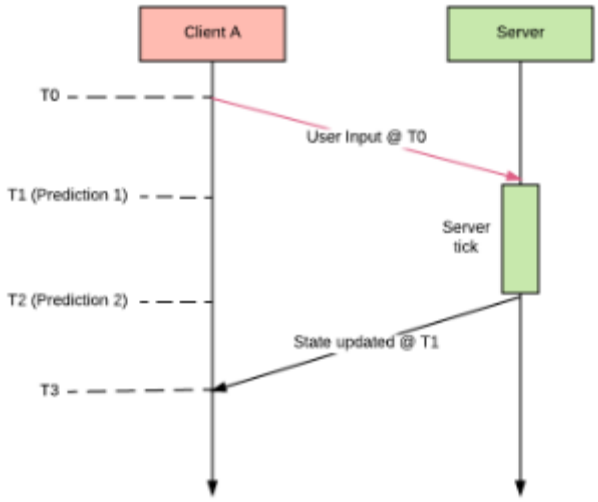

Figure 10: Client prediction in a Client-Server Model (Lim Q W 2017).

Client prediction only functions if the update logic of the game state is deterministic – no randomness so that the server and client produce the same game state using the same inputs, and only if clients have all the required information to run the logic of the game. There are advanced models that can function with those two requirements not being met (Lim Q W 2017).

More of the game's logic would be handled on the client's side and there must be some solution to the conflict between the client's predicted state and the actual state received from the server. Prediction through this method allows for responsive local controls for every client, due to less input latency, however without interpolation/extrapolation there the game will still have noticeable delays (Lim Q W 2017).

Clients can smoothly interpolate their state between updates received from the server rather than just relying on their positions sent from the server. Due to the entity always being interpolated according to the most recent update received, the state of that entity will always be slightly behind the actual state in time. Client interpolation through this method allows for smooth motion and movements of other clients whilst maintaining as low of latency as possible, however without prediction the game can still feel sluggish (Lim Q W 2017).

On the other hand, client extrapolation (known as deck reckoning) is a technique which involves predicting an entity's future position, rotation, velocity and other data based on its last known values by estimating where the entity will be the next number of frames until the correct state arrives. The longer the guess is in the future, the more likely it is to make an error. This technique reduces the packet delays, allowing for interactions through real-time to be much faster and less delayed. It deals with lost and/or missing packets more effectively when there are many players. This method is not as precise as client interpolation and extrapolation may cause variations in what every client sees (MY.GAMES 2023).

One technique to improve the fairness between players who are connecting at various speeds is lag compensation. Whenever players interact with each other, the server rewinds time to compensate for the latency

of player that started the interaction to see if the interaction would've landed on the other player based on where the other player appeared on their screen at the time they started to interact. This is very crucial as by the time the input reaches the server, without lag compensation, this time would have allowed the other player to move elsewhere and in turn not register the interaction (MY.GAMES 2023).

Lag compensation makes the game feel much more responsive and fairer even with network delays, however this is viewed as being controversial as there is a susceptibility to cheating. If the server trusts the timestamps sent from the clients, a client can trick the server by sending a later shot and faking that it was performed before the actual time it was made and for this reason lag compensation should be avoided if possible. Using client prediction, interpolation/extrapolation and dead reckoning does not imply any trust in clients (MY.GAMES 2023).

2.3 Security & Integrity in Multiplayer Games

Multiplayer games are heavily exposed to a multitude of security threats, including packet sniffing, distributed denial of service (DDoS) attacks, man-in-the-middle attacks and in-game cheating. The choice of architecture determines how vulnerable the networks are to these vulnerabilities, for example P2P architecture is particularly vulnerable to direct attacks and IP exposure. Even minor security breaches in multiplayer games can result in mass cheating or breaches in private player data, undermining the game's reputation and player trust (Siddiqui L 2024).

Within server-authoritative architecture, the single source of truth is the server and is responsible for validating every client's actions to prevent unauthorized changes in the game state. Therefore, this approach is utilized in commercial multiplayer games to ensure cheating is minimized and fair play is maintained (Valve Corporation 2024). In contrast, client-authoritative topologies are easier to incorporate but are much more susceptible to every security threat as the connections between clients are direct and each act as the source of truth which also leads to clients easily being able to manipulate the state. As a result, P2P models are avoided for competitive games (MY.GAMES 2023).

A basic anti-cheat via the client-server architecture and validation checks is one option for preventing clients from cheating. The server would act as the authoritative truth and by making it never trust the client fully, performing validation on player movements, positions, actions and changes in their state ensures that every client is playing within the server's rules (Valve Corporation 2024; MY.GAMES 2023).

Simple checks like verifying player movement speeds to make sure they are not moving faster than allowed or performing an inhuman number of inputs. Whilst these server-authoritative checks are effective, a lack of optimization will increase server load and potential latency. Furthermore, basic validation can be bypassed by more sophisticated cheats which may only be detected via advanced detection methods, such as machine learning or behavioural analysis. However, this would also increase server load and raise privacy concerns (Valve Corporation 2024; MY.GAMES 2023).

One simple method to secure the connection between clients and servers is through encryption of data within the commutation channel to prevent interference of the data packets that are being shared. Main protocols for encryption are through TLS/SSL protocols. The game can also implement authentication in which clients must prove their identity before joining to shield the network against spoofing. (MY.GAMES 2023).

2.4 Social, Legal & Ethical Considerations for Online Multiplayer

Multiplayer games must comply with data protection acts like the General Data Protection Regulation (GDPR) and developers are entirely responsible for keeping player data secure from unauthorised access while establishing a positive online community through moderation and reporting tools. Failing this, legal liability and harm in reputation is inevitable.

Maintaining fair play is also an ethical imperative, not just a technical challenge to overcome. Players in games can behave disruptively in forms of scamming, hacking, cheating, betraying, insulting, deceiving and harassing each other. These behaviours group together under the act of 'griefing' – purposefully disrupting the gameplay and other players enjoyment (Sparrow et al 2020).

Multiplayer games must understand the concept and framework of 'ludomorality' (Sparrow et al 2020): the key themes when evaluating acts in games ethically which highlights the importance of meaningful consequences, clear rules and respect for player sensibilities and virtual experiences. These four themes shown in Appendix Figure A.2 are the:

- boundaries within the game – the rules defined in-game that determine the actions possible
- consequences for play - the rewards or penalties based on how players interact in the game world
- player sensibilities – how players are influenced emotionally and ethically through events in-game.
- and virtuality – how convinced players are of the simulated game world to allow escapism and engaging experiences (Sparrow et al 2020).

2.5 Unity's Tools & Services

Unity's NGO is a high-level library built for abstract networking logic. The data and objects within a game can be transmitted and synchronized across a networking session to players easily without focusing on low-level protocols and other frameworks. This framework includes two data syncing mechanisms – RPCs for data only useful for a short while when it's gathered, and NetworkVariables for persistent states and data that will be useful for much longer. These both send information over the network and between clients and the server. It is crucial to decide on the most suitable type as the invalid option can cause bugs, increase code complexity and use more bandwidth than essential (Unity Technologies 2025f).

The benefits of this library and other game engine networking solutions such as Mirror and Photon is that it is already established in Unity's ecosystem – it functions with Unity's network transport and other services efficiently and seamlessly (Unity Technologies 2025b). Mirror is an open-source alternative and provides more in-depth features but is slightly less integrated within Unity (Mirror Networking 2024). Another option is Photon, a third-party, cloud-based framework that is easy to use but can be costly with bigger games (Photon 2025).

Unity's Network Simulator tool allows artificial simulation of various network conditions by allowing configuration of network parameters of the ping and packet loss of clients as well as network events such as disconnects and lag spikes. This allows testing of multiplayer networks in different, suboptimal connection conditions (Unity Technologies 2025e).

Unity's Authentication service provides a way to identify players so security, consistency and safety is maintained for every interaction within the network. Additional features, techniques and services can be implemented on top of this identification, such as giving players names and setting up lobbies. Cross-platform account and authentication is offered with this service to support cross-play and progression with customizable sign-in methods (Unity Technologies 2025c).

Unity's Lobby service allows for players to find and connect each other online via lobbies. This is carried out by players being able to search for available game sessions and choose one to join or via the host of the lobby sharing a lobby code if the lobby is set to private. This service is very beneficial to enable players to easily and quickly find each other and agree to start a game (Unity Technologies 2025d).

Unity's Relay service ,previously mentioned in more detail, allows for further multiplayer connectivity via communication through relay servers without dedicated game servers. Common issues that occur during connection such as changing networks and IP addresses, NAT and firewall restrictions can be bypassed by players connecting to a public endpoint that is more reachable than connecting directly to a host's IP address (Unity Technologies 2025a).

2.5 Summary & Gaps in Research

2.5.1 Summary & Limitations of Key Findings

The thorough literature review reveals that real-time multiplayer games demand efficient and robust solutions for key technological challenges, namely state synchronization, latency mitigation and security. There are many decisions for a designer to make, for instance the architecture where a client-server topology is the industry standard due to its scalability, security and centralized authority, but are not without challenges.

Complex techniques for example client prediction and lag compensation are necessary for a smooth multiplayer experience, more so in physics-based games. Developers benefit from already established tools and services,

such as the suite of networking tools and services Unity provides that are integrated into the game engine. Most of these solutions still face limitations regarding responsiveness in real-time, reconnection and scalable anti-cheat for small-scale games.

Although the impressive advancement made in the multiplayer games industry, this existing game engine and the frameworks it provides are still incomplete and lacking functionality and network resilience – seamless reconnection, host migration and state restoration after clients or the host disconnect are not supported. The tough challenge of physics-based state synchronization with minimal delay must still be dealt with, as well as the limited anti-cheat and security features provided by Unity. Another lacking issue to undertake is the insufficient handling of network instability and lack of dynamic latency/jitter configuration based on connection stability.

2.5.2 Position of this Project

This project will address these gaps in solutions by focusing on:

- Incorporating comprehensive and reliable state synchronization and reconnection mechanisms custom-made for small group, real-time, physics-based games in Unity
- Utilizing the server-authoritative topology and logic along with modular managers to prevent cheating and variable manipulation while ensuring fair play
- Taking advantage of Unity's latest networking tools and services to streamline connection, authentication and NAT traversal

2.5.3 Justification for Approach

The chosen approach of combining Unity's NGO, Authentication, Lobby, Relay and robust server-side validation offers a scalable, practical and secure solution for the solutions identified. Unlike large-scale MMO and local coop/LAN games, this solution is specifically tailored and optimized for two or more players (but not up to player counts typical in MMOs) in physics-based and real-time environments over the internet. This ensures suitability to the projects aims and the advancement of multiplayer networking while maintaining network efficiency.

Chapter 3: Methodology

3.1 Requirements Specification & Risk Assessment

In this section, the design principles for the multiplayer game network are defined to guarantee the aims of the project are met.

This multiplayer game network must support real-time interactions between two or more players and provide a seamless multiplayer experience conducted through several integrated systems. To reach this solution, certain features and systems must be implemented. Firstly, a lobby and matchmaking system with roles assigned to the host and clients with a player readiness indication, all made secure through unique player authentication.

The real-time synchronization of the positions and physics states of players, game events and scores forms the network architecture's foundation, while server-authoritative logic manages crucial in-game actions and events that remain secure and fair. A consistent experience for each player despite connection problems will be handled by the system incorporating reconnection and state recovery for players that disconnect, reintegrating them seamlessly back into the ongoing game.

The reliability of the network is further enhanced by the monitoring of network performance metrics and each client's ping, with varying connection quality being catered for by dynamic parameter tuning of the advanced network techniques used for latency mitigation. Security is maintained utilizing server-authoritative design for validation and integrity. A dedicated network stress testing mode must be incorporated for evaluating the solution's performance under various conditions to reveal whether it can handle realistic usage scenarios with fairness, responsiveness and reliability.

Low latency performance requires prioritization for this system – clients must have a ping below a target of 100ms average for responsive gameplay and state updates, even under suboptimal connection conditions. Furthermore, the solution must remain reliable functionality during these suboptimal connections.

Another key requirement is security upheld by the network architecture, protocols and authentication to protect player data and halt unauthorized access. Scalability is required to allow for expansion in the future without major restructuring of the solution. Compliance with the legal and ethical standards is mandatory, with adherence to GDPR, licensing of third-party assets and a code of conduct enforced to players.

Regarding the interface, a UI must be implemented to support all functionality and navigation including authentication, lobby management and gameplay along with a real-time display of connection stability. User experience must be improved via game elements such as game scores, countdown and list of lobbies/players. These elements must be updated across the host and clients by reliable messaging transmitted through a rUDP transport protocol to ensure they are delivered successfully.

This solution assumes that two or more players are involved during operation – the network architecture must be optimized and gameplay balanced for this scale specifically, therefore one constraint is the lack of a single-player mode or any AI opponents. Development is constrained to the Unity engine, which utilized the C# programming language and the NGO framework.

The inherit limitations and networking capabilities that come with this game engine must be accepted. The solution does assume that every player has a stable connection but mitigation strategies for poor connections must be used. Additionally, scope limitations have been set - virtual reality and augmented reality are not implemented; user interface is minimal and designed for functionality rather than aesthetic polish; and commercial monetization systems like in-game purchases are not incorporated whatsoever.

These requirements lay out a clear blueprint for the design and implementation of the solution, assuring all aspects deemed essential of multiplayer networking, security, usability and compliance are dealt with. A risk assessment and mitigation plan is essential and is displayed in Table 2.

Table 2: Risk Assessment and Mitigation Plan.

Risk	**Likelihood**	**Impact**	**Mitigation Plan**
Network failure during demo	Low	High	Make local backup video demo
Unity service outage	Low	High	Make local backup video, test with local server
Data loss	Low	High	Use version control, regular backups, test demo prior to demonstration date
GDPR or data privacy breach	Low	High	Avoid personal data, use anonymous identification
Cheating, exploits and spoofing	Medium	Medium	Sever-authoritative logic, validation checks

The subsequent section analyses the architecture and design choices made for the system to fulfil these requirements, as well as the implementation pipeline.

3.2 Analysis of Methodological Approach

3.2.1 Methodology & Development Approach

The methodology adopted in this project is a modular manager-based approach that follows OOP principles to implement a real-time, physics-based multiplayer networked game within the Unity engine. The design is built on best practices discovered in the multiplayer networking field (as explored in Chapter 2) to leverage Unity's built in frameworks and additional services while addressing the key challenges identified in the problem statement.

UDP is the underlying protocol to facilitate low-latency, consistent and high-frequency state synchronization of data that changes constantly and can sacrifice potential data loss, with an rUDP layer implemented at the application layer by Unity's NGO to improve message reliability for crucial data which must be delivered.

For the architecture, this solution follows a client-server model where critical game logic and validation are enforced on the host player, ensuring fairness and minimizing cheating. Unity's integrated services (Authentication, Lobby and Relay) are employed to achieve secure matchmaking, connectivity and NAT traversal while the core functionality of the gameplay and network are handled via singleton managers that are encapsulated within their own class to separate concerns, promote code reusability, scalability and maintainability.

The singleton pattern ensures a single point of access and consistent state management across scenes. Overall, this methodological approach intends to address the key challenges of synchronization, network resilience and security suited to two or more players in real-time, physics-based multiplayer games.

A systematic, iterative approach was followed for the development – each stage progressively implemented functionality and validated the core networking elements. This methodical process addressed fundamental networking challenges sequentially with each prototype enhancing functionality from prior iterations.

3.2.2 Technology Stack & Justification

Each component within the technology stack was carefully chosen based on its suitability and capability to address the challenges specified in the literature review and problem statement. Balancing efficiency with the networking performance requirements in real-time, physics-based multiplayer games was essential.

The Unity game engine was selected as the development platform for many compelling reasons. For one, Unity has an integrated physics simulation system which is required for the solution's physics-based gameplay. Another reason was for the availability of documentation and community support which provided resources for implementation guidance and troubleshooting.

Additionally, the component-based architecture Unity employs aligns well with the modular approach a complex networked game needs. The visual scene editing combined with code-based development further accelerates the prototyping process.

After evaluating various networking solutions including Mirror and Photon, Unity NGO was decided upon for the advantages it comes with. One major factor was the tight integration within Unity for seamless incorporation with Unity's other services and components. This framework is also built to support server-authoritative design which is required for the security and anti-cheat measures this solution seeks.

Furthermore, this framework is flexible with support for UDP and extra reliability features via rUDP. Finally, this is Unity's official networking solution so it receives consistent updates and improvements, making this very appealing.

Unity's services were chosen to be integrated to address the requirements of this solution. Unity's Authentication service for secure player identification, persistent player data, basic security and reconnection. Unity's Lobby service facilitates matchmaking and lobby creation, bypassing the need for any custom implementation. Unity's Relay service solves NAT traversal issues, improves security by obscuring client IP addresses and allows connections across different networks without port forwarding.

UDP was picked as the underlying transport protocol along with a reliable layer of rUDP. This accommodates the requirements of low-latency for responsive real-time, physics-based interactions while allowing reliability only where necessary – game state changes, scoring events – while keeping frequently changing updates lightweight. Unlike TCP's strict delivery, UDP handles unstable network conditions effectively and enables smooth physics synchronization despite suboptimal connection quality.

A modular, manager-based architecture was designed to leverage OOP principles of encapsulation of functionality, singleton patterns for consistent global access, loose coupling, scalability and maintainability. Efficient state synchronization was achieved using Unity's Network Variables for critical game state that requires continuous synchronization when it is altered. Additionally, RPCs were used for event-based communication and actions requiring validation from the server and custom serialization to minimize bandwidth usage and optimize transference of complex data structures.

3.2.3 Development Timeline & Implementation Pipeline

The development timeline is set out in Figure X that includes the project's various stages, split into research, analysis/design, prototyping and testing/validation. The columns show the weeks that these stages were completed in.

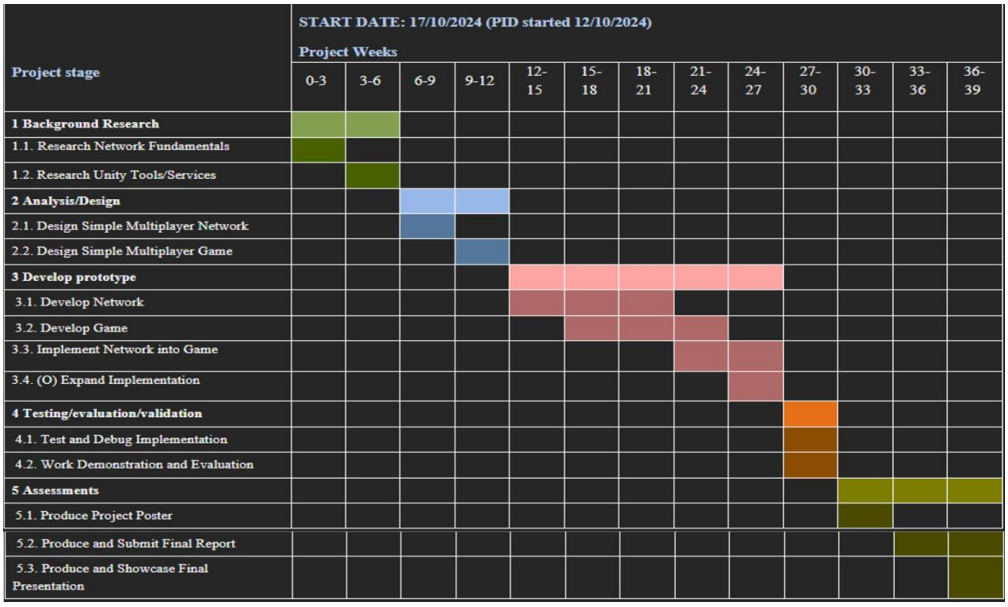

Figure 11: Project Development Timeline.

Regarding the implementation pipeline, the initial phase was entirely focused on laying the foundation for the fundamental networking infrastructure with no concern for advanced networking techniques. Basic authentication for unique player identification was achieved by making use of Unity's Authentication service with simple player objects limited to rudimentary movement that was synchronized via a minimal network transport supported by Unity's NGO.

Basic communication was handled through a client-server model where player objects and their positions were synchronized between clients and the host player. To test this prototype, a simple 2D environment based on Pong was developed – each player was a paddle with the goal of hitting a ball object into the other player's 'goal' (an invisible wall which checked for collisions with the ball object).

This established a minimal viable product in which multiple clients could connect to a host and transfer basic state information; however, movement was calculated on each client individually. Complexity was minimal with the 2D environment (visualized in Figure 12) and limited interactions between players which allowed for a clear observation of network behaviour without complicated 3D physics and feedback on connection quality and basic synchronization abilities. Once basic connectivity was proven to be reliable, the process moved to build on more elaborate state management, synchronization and features.

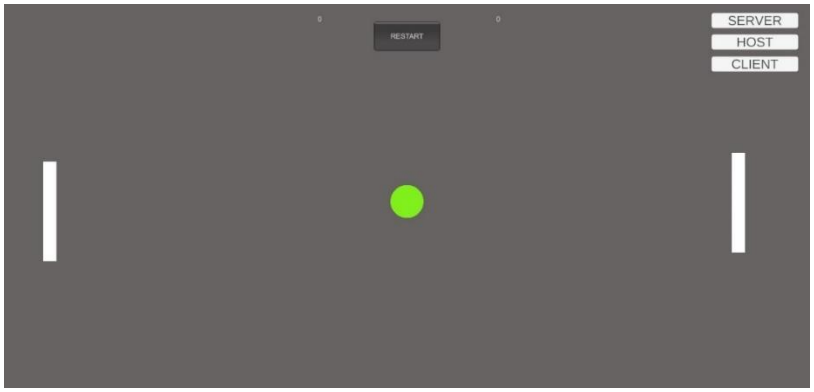

Figure 12: First Prototype in Gameplay View.

The second phase transitioned into a 3D environment, adding complexity in position and rotation synchronization as a whole axis was introduced for both (see Figure 13). On top of this, basic physics were implemented with interactions based on these physics could occur between players. This demanded a refined usage of Network Variables for more efficient state management as more data needed to be exchanged in the network.

Server and client RPCs were developed to establish server-authoritative logic for critical game events – player movement was now handled by the host player which acted as the source of truth. The basic lobby system was implemented for lobby creation and joining functionality. This phase represented a significant rise in complexity and allowed for further testing of the architecture's capability to handle sophisticated data synchronization while validating the solution's scalability as complexity increased.

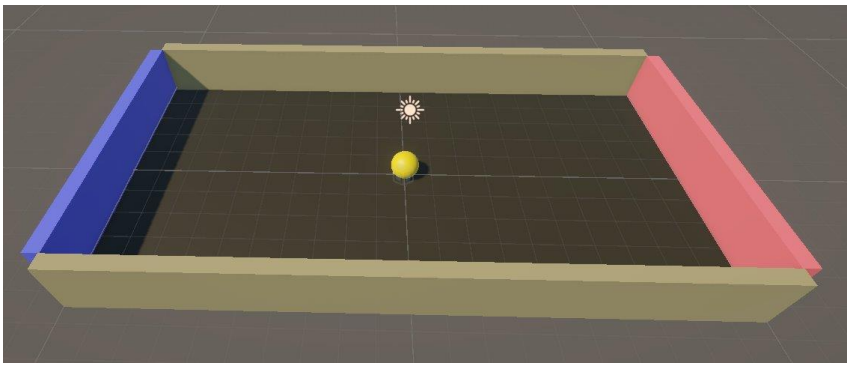

Figure 13: Second Prototype in Gameplay View.

Phase three concentrated on dealing with core multiplayer challenges with advanced networking techniques. The previous iteration faced a major lack in responsiveness for the client's movement – significant input delay as well as minor desynchronization of physics states was prevalent. To combat this, client-side prediction together with server reconciliation was brought into play to reduce the perceived delay of inputs and correct diverging player positions. This further tightened security as server-authoritative validation was more extensive.

Network traffic was optimized with refined custom data serialization and transmission frequency, improving overall network performance. A separate scene for stress testing the network was integrated to test whether this was the result. The advanced network techniques were augmented through thorough testing with artificially induced latency and packet loss through Unity's Network Simulator tool to fine tune parameters on realistic conditions.

The final phase developed focused on optimization and extending existing features alongside the UI. Smooth interpolation was developed for player character movements to increase responsiveness. Furthermore, the lobby system evolved to include player ready status, lobby search and relay integration – previous prototypes had players host on their own IP address.

Every client's ping was calculated and displayed to every player in session with high-latency clients being shown feedback through a warning about their unstable connection. Also, the UI received a major update aesthetically. Dynamic parameter adjustment based on connection quality was implemented to improve network resilience, as well as reconnection and state recovery functionality allowing for disconnected players to reconnect.

The following sections elaborate on the system architecture and design, implementation of the network, security mechanisms and test strategies used to realize the project's aims.

3.3 Design

3.3.1 High-Level System Architecture Design

The high-level system architecture has been designed specifically for this project's use case within Unity and operates on a modular, manager-based pattern where each core functionality is encapsulated within its own dedicated manager class. This approach also makes use of OOP principles and the singleton pattern to establish consistency within state management and ease of access across scenes and network sessions.

At the core of this system is the CoreManager, which acts as the central manager for maintaining references to other scripts and initialization for Unity's services. Upon the application's startup, the CoreManager is responsible for all managers required in the current scene are instantiated and made accessible as singletons.

This guarantees that the other managers essential for their respected functionality have only one instance that can exist at any time and, if appropriate, persists across scene transitions. The flow between the networking layer (via NGO) Unity services and other gameplay and UI managers is coordinated through the CoreManager.

Every other manager has its own crucial responsibility:

- LobbyManager deals with all lobby functionality: creation, searching for lobbies, joining, player management within a lobby and leaving lobbies, interacting with Unity's Lobby and Relay.'

- PingManager monitors latency for players within the network with UI updating to display ping values.
- ReconnectManager manages the recovery and persistence of the game state for disconnected players.
- MenuManager handles authentication and transitions between menu and game scenes.
- GameManager oversees gameplay functionality: game state, scoring and transitions from the game scene.
- MenuUIManager & GameUIManager for displaying and updating the respective UI elements.
- NetworkStressTestManager for stress testing the network.
- LogManager to output and store debug logs in a local text file.

Some scripts are not made as singletons and are instantiated per player or lobby item as required:

- PlayerNetwork for managing individual player character movement and collision alongside client prediction and interpolation techniques.
- PlayerListItem for displaying individual player UI elements in the joined lobby's player list.
- LobbyListItem for displaying individual lobby UI elements in the lobby search results.

The managers communicate with each other via direct referencing (set by CoreManager) and suitable event-driven patterns, establishing OOP principles with somewhat loose-coupling, modularity, a separation of concerns and encapsulation. This allows for easy extension of new managers or features without existing functionality to the core logic in the networking or gameplay being disrupted. The overall instantiation and interactivity of and between managers and services is displayed in Figure 14.

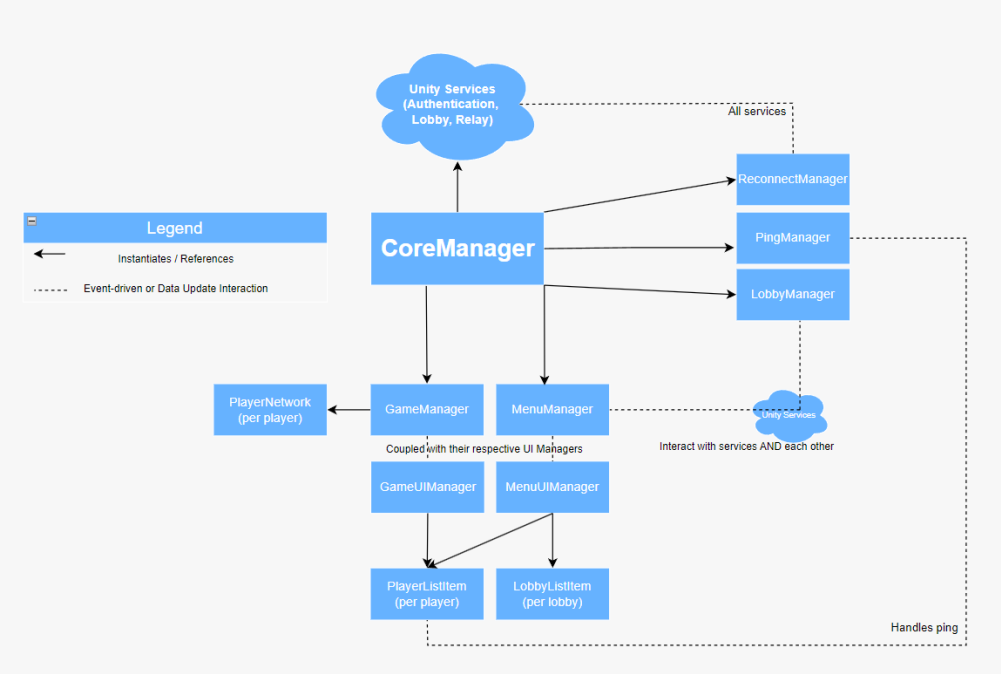

Figure 14: Flowchart for Manager and Service Instantiation / Interactivity.

Figure 14 illustrates this modular manager-based architecture for the system - managers interact mainly through direct references and event driven patterns, organized and coordinated by the CoreManager and respective Unity service.

Functionality that is per player/lobby is managed by the GameManager (PlayerNetwork for each networked player object), GameUIManager (PlayerListItem for each player in the lobby) and MenuUIManager (PlayerListItem, LobbyListItem for each lobby found from searching through Unity's Lobby service).

Overall, this design for the system architecture is suitable as it provides a blueprint alongside a visualisation of how all the manager's work together and with Unity's services. Also, it reinforces the solution with maintainability and extendibility to support improvement or expansion in the future. The next section will feature a deep dive into the design of the network architecture.

3.3.2 Network Architecture Design

The network architecture for this solution is built on a client-server model using Unity's NGO to establish sever-authority with the host player being the authoritative server. All pivotal game logic, including player movement, physics simulation, scoring and stage changes is executed and validated on the server, eliminating the risk of cheating severely and maintaining a consistent game state between all clients.

At the core of this networking system is Unity's NetworkManager which behaves as a central hub for all netcode connectivity and functionality. The NetworkManager uses the Unity Transport (UTP) layer, built around UDP with NGO implementing an rUDP layer at the application layer. Low-latency, highly frequent data transmission is carried out through the UDP layer while important events that require reliability are transmitted via the rUDP layer. The CoreManager script confirms that the NetworkManager is instantiated and persists across scenes while preventing duplicate instances and maintaining consistent network state.

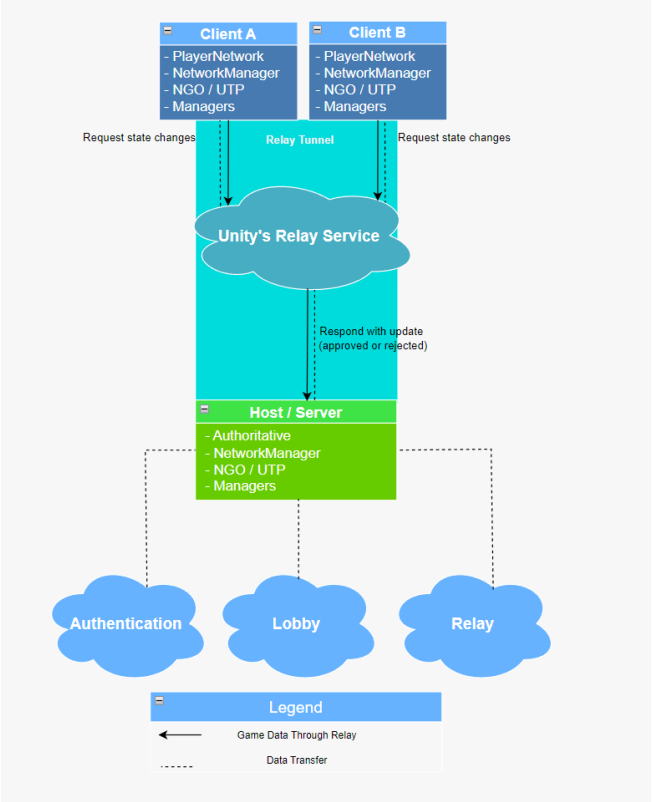

Figure 15: Network Architecture and Data Flow for Two Player Multiplayer.

Figure 15 portrays this design for the multiplayer network and data flow. Clients connect to the server through a secure relay tunnel – no direct connections between clients are established and all traffic within the network is securely routed. The server acts as the authority and validator of all game logic and state changes, handling all requests from clients whilst interacting with Unity's services for secure login, matchmaking and connectivity. This ensures all data critical to the gameplay is validated centrally by the server which additionally interacts with Unity's services.

The step-by-step process from the initial player authentication to when gameplay begins is displayed in Figure 16 which represents how a player first signs in, is authenticated and assigned an authId from Unity's Authentication. From here, the player either hosts a lobby or joins an existing lobby, requesting lobby details from Unity's Lobby service.

Upon joining or creating a lobby, the NetworkManager creates a connection with the relay allocation from Unity's Relay service and MenuManager's PlayerIds are mapping to AuthIds within LobbyManager. All players now ready up and then proceed to start the game – if all players are readied up then the LobbyManager starts the game by loading the gameplay scene. Here is where the ID mappings are shared to the GameManager, which responds back with the gameplay functionality to the player.

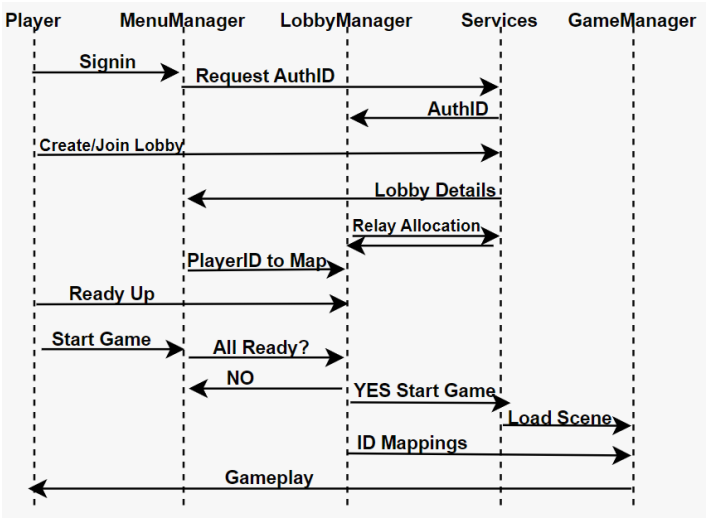

Figure 16: Sequence Diagram for Player Authentication, Lobby Initiation and Game Start.

Figure 16 also highlights how vital the integration and coordination of local script managers and Unity's services are to result a secure and seamless multiplayer system. When players agree to start the game, the NetworkManager instantiated a player prefab object as a NetworkObject, ensuring each player has a unique network identity and character.

3.4 Implementations

3.4.1 Modular Manager Architecture Implementation

The modular, manager-based design that has been developed orchestrates the user flow of interactivity through the multiplayer system by dividing responsibilities between manager classes with each dedicated manager being handled and referenced via the CoreManager singleton (see Listing 1).

```
// If theres an instance already which is not this one (likely as this was loaded in new scene)
if (Instance != null && Instance != this)  {
        // destroy this instance, stop initialization
} Else {
        Instance = this; // Else, there is no other instance so set this as the global instance
        DontDestroyOnLoad(gameObject);  // make it persistent across scenes
         // ... Initialize Services and Managers
```

Listing 1: CoreManager Initialization, Singleton Pattern Setup

Upon application startup, the CoreManager not only initializes Unity Services – required for Unity service use – and the NetworkManager but also gets or instantiates the required managers and sets up references for them so that the other managers can reference each other from the same single consistent instance. Some scripts that are not persistent across scenes are also considered by the CoreManager by checking the current scene and setting up the corresponding managers. Firstly, the Lobby scene's sign-in screen is shown where the MenuManager handles user authentication using Unity Authentication, as shown in Appendix Listing A.5.

After players authenticate themselves successfully, the user proceeds to lobby management. Here, the LobbyManager allows for lobby creation, search and joining utilising Unity's Lobby and Relay services. Once a player that is seeking to host creates a lobby, a relay allocation is generated and used for secure client-server connectivity. Now players can join the lobby and indicate their readiness – the LobbyManager maintains and synchronizes ready status for each player through Server and ClientRPCs to propagate state changes, seen in Appendix Listing A.7. The host can transition the lobby into the game scene once all players are ready. This is managed by the MenuManager which invokes a ServerRPC, triggering a synchronized scene load for every client (see Appendix Listing A.8).

Upon entering the game scene, the GameManager comes into play through the CoreManager's scene loaded event where once again the CoreManager sets up the necessary managers depending on the scene (see Appendix Listing A.1). The managers interact by referencing each other through the CoreManager, establishing tight coordination and loose coupling.

Each aspect of the multiplayer flow, from initial authentication and lobby management to gameplay, is handled by specialized components carefully coordinated by the solution's modular approach. The concerns are separated clearly, while the direct referencing pattern minimizes coupling and promotes persistent resilience across scene transitions.

3.4.2 Network Architecture Implementation

Crucial game data such as player movement and game scoring are only updated by the server, following the server-authoritative logic. These updates are transmitted to the host through NetworkVariables and RPCs, handled in the GameManager script (see Appendix Listing A.2).

Bandwidth is optimized by NGO's NetworkVariables which converts data objects into a data structure into a byte stream for storage, transfer and distribution, also known as custom data serialization, as shown in the PlayerNetwork script (see Appendix Listing A.3). It also sets read and write permissions to either owners of the specific data type or everyone. Further optimization is achieved by only sharing data across the network that has

changed. This is better for network performance compared to constantly sharing variables as repeatedly sharing data that is the same results unnecessary bandwidth usage.

Synchronization of player movement and physics states is handled by a combination of client prediction, server reconciliation and server-authoritative updates. The server receives requests for movement from clients via Server RPCs and sends the state update after these movements have been calculated back to all clients to update their positions and physics states, depicted in Appendix Listing A.4.

After the server performs these updates, they are transferred back to the clients via a ClientRPC which synchronizes clients to the same positions and physics states the servers truthful game state is at, illustrated in Appendix Listing A1.6. Every NetworkObject is registered in the NetworkManager's prefab list visualized in Figure 17 – only these objects can be spawned and synchronized over the network through RPCs. These NetworkObjects include the Player and Ball objects, as well as all the objects containing managers that utilise RPC's.

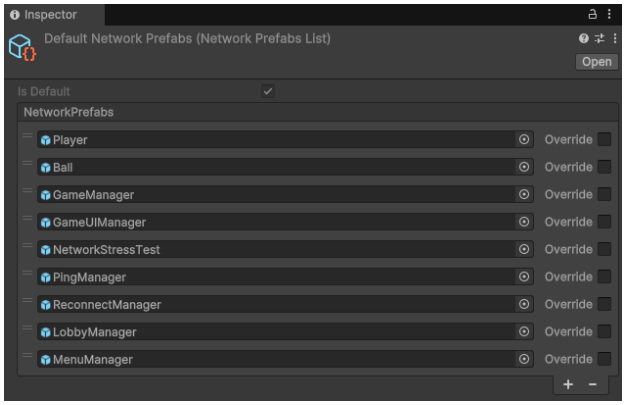

Figure 17: NetworkManager's Network Prefabs List in Unity.

Overall network health is monitored using the PingManager script where upon the RTT is determined for each client and providing feedback via the UI displaying ping accordingly (see Listing 2). Network resilience and reconnection techniques are outlined in section 3.4.5.

```
// Loop ran every 2 seconds

if (NetworkManager.Singleton.IsClient) { // if this is a client (NOT RAN ON SERVER)

        float sendTime = Time.realtimeSinceStartup; // record current time

        string playerId = AuthenticationService.Instance.PlayerId; // get playerID

        RequestPingEchoServerRpc(sendTime, playerId); } // send ping request to server

        // calculate RTT based on response, update ping display
```

Listing 2: PingManager Measuring RTT for Monitoring Network Stability.

3.4.3 Client-side prediction & Reconciliation Implementation

Responsive gameplay and controls despite network latency is a major challenge in physics-based, real-time multiplayer games. To mitigate this, the solution deploys advanced networking techniques, namely client-side prediction and server reconciliation to sustain immediate feedback for players during which consistency is maintained across all clients.

Client-side prediction enables players to visualize immediate results of their inputs without the delay waiting for the server to confirm, calculate and transmit back the result. This heavily reduces input delay and provides a responsive local experience even during network delays. This is highlighted in the PlayerNetwork class's FixedUpdate() method, which is called at a fixed interval independent of the frame rate which ensures consistent behaviour no matter what hardware the clients are using (see Listing 3).

Physics forces are applied locally in an instant before the input is sent to the server for authoritative validation, creating the illusion of zero latency for clients as movement is perceived instantaneously.

```
// In fixed update, ran at a fixed interval
if (!IsOwner) return; // Only owners of their own player object run this
        // Apply movement and physics locally first for responsive input
        Vector3 force = InputKey * acceleration;
        rb.AddForce(force, ForceMode.Force);
        RequestAccelerateServerRpc(InputKey); } // Then send to server
// do same for deceleration
// PlayerNetwork – ServerRPC for reconciliation
if (!IsOwner) return;
        Vector3 posDifference = serverPos - transform.position; // calculate position difference in server's and
        client's state
// if position is larger than threshold
        // set as correcting position, reconcile the position
```

Listing 3: PlayerNetwork – Client-Side Prediction and Server Reconciliation ClientRPC.

Responsive gameplay is attained through client-side prediction; however, this technique can easily lead to diverging states between the game state predicted by the client and the server's authoritative state. To deal with

this issue, a reconciliation system has been fitted to make periodical checks for significant differences and smoothly corrects them to the truthful state on the server (see Appendix Listing A.8).

After the clients retrieve the authoritative position and velocity of their player object from the server, the values of which are compared to the client's own predicted state. Only if the difference exceeds a configurable threshold is a correction applied to update the client to the server's state (see Appendix Listing A.9).

Rather than snapping the player character to the correct position and velocity which would be visually jarring, a smooth interpolation is applied over a short period of time, seen in Appendix Listing A.10. This approach is appropriate as corrections maintain visual clarity and smoothness while eventually aligning the client's state to the server's truthful state.

Due to the conditions of the network and client's connectivity fluctuating from client to client, the system needed to adhere for higher latency clients. The parameters of reconciliation are therefore tuned dynamically based on each client's ping – the reconciliation interval value determines how frequently clients request the server's authoritative state and the position error threshold dictates by how much the predicted state can differ before applying a correction.

A slightly higher threshold and less frequent reconciliation improve high-latency clients' experience as the delay will be less noticeable due to less frequent corrections, while stricter thresholds are more effective for low-latency clients. For non-owner players (other clients' characters visible to the local player), an alternative technique is required instead of prediction. These player objects are interpolated smoothly between network states (see Appendix Listing A.11) which shows smooth movement for all the other players' characters even whilst network updates are irregular.

Thorough testing was essential for these advanced network techniques to operate as intended – a balance of the parameters, for say the reconciliation threshold, was a difficulty as if set too low, minute errors in the prediction results in jittery movement as corrections are too frequent. If set too high, major divergence occurs before the client's state is corrected. The solution finalized on utilizing a dynamic threshold that adapts to the client's measured network performance to avoid these issues and provide a responsive experience for varying degrees of network latency.

3.4.4 Unity Services Integration

A robust multiplayer network insists on each player having their own unique and persistent identity. As briefly discussed previously, Unity's Authentication service supports this capability by enabling secure sign-in, tracking of users and association of network actions with verified players. This lays the foundation for additional Unity services such as Lobby, Relay and implementations of the reconnection system and anti-cheat mechanisms.

Authentication is initiated by the MenuManager where each player signs in with an inputted display name (see Appendix Listing A.5) and are granted a unique PlayerId used for matchmaking, lobby and game session functionalities. These are logged for debugging whilst giving feedback through the UI/UX – users see a clear sign-in page with an input box and then upon sign-in are transferred to the next screen with their display name visible at the top.

Generated PlayerIds are used throughout the network flow, for example when a player hosts or joins a lobby their identity is tracked via this ID to ensure consistent association even after disconnection and reconnection. Player management is an additional benefit of this approach due to the mapping between network ClientIds and authentication PlayerIds being managed by the GameManager portrayed in Appendix Listing A.12.

This mapping is key for advanced features such as persistent player data and reconnection. When a player disconnects, the ReconnectManager class uses their authentication ID to save their state and, upon reconnection, restore their state just as they left it. This is shown in Listing 4. Henceforth, authentication proves vital for this solution's security, integrity and seamlessness. The pitfalls of anonymous or spoofable network identities are avoid whilst supporting matchmaking, session management and state recovery across the multiplayer lifecycle.

```
// Recording Disconnect

disconnectedPlayers[authId] = DateTime.Now; // hold time of disconnection

// ServerRPC for reconnection

// .... verify grace period, load saved state

RestorePlayerStateClientRpc(authId,restoredState.position,restoredState.velocity,  restoredState.angularVelocity, restoredState.score);} // Server restores state of client object to client
```

Listing 4: ReconnectManager storing player disconnects through authentication IDs and then reconnection via a ServerRPC.

Unity's lobby service is used for player/lobby discovery, lobby creation and matchmaking. Players can create a lobby which clients are able to join via the shareable lobby code given to the host or through a list of lobbies found online through a search function. Alongside these capabilities, ready statuses (see Appendix Listing A.13) and host/client roles (see Appendix Listing A.14) are managed by the Lobby Manager and PlayerListItem.

Players can host on their local IP address, or via Unity's Relay Service which allows communication using relay servers and not actually connecting directly. The relay server will not only bypass firewall and network address

translation (NAT) restrictions (common issues during connection) but reinforce security as host/clients never connect directly, thus keeping every client's IP address anonymous (see Listing 5 and Appendix Listing A.16).

```
// Method to create lobby

string lobbyName = playerName + "'s Lobby"; int maxPlayers = 4; // set lobby name

Allocation allocation = await RelayService.Instance.CreateAllocationAsync(maxPlayers); // Create a Relay allocation

// ...Get the join code for clients to use from allocation

var relayServerEndpoint = allocation.ServerEndpoints[0]; // (RELAY) Get the Relay server's endpoint

// ...(RELAY) set IP and port as the server's endpoint  and configure transport with these

CreateLobbyOptions createLobbyOptions = new CreateLobbyOptions { // Create Lobby with relay join code

// ... (RELAY) public, set host and store this code instead of IP/port

Lobby lobby = await LobbyService.Instance.CreateLobbyAsync(lobbyName, maxPlayers, createLobbyOptions); // (LOBBY) Create the lobby

// ... (NGO) Start as host
```

Listing 5: LobbyManager's create lobby function with a relay server allocation.

When a player seeking to be a host creates a lobby, the system allocates a Relay server and generates a join code (stored in the lobby's metadata). Players can join either through a lobby code (see Appendix Listing A.14) shared manually by the host player or through the lobby search function which gathers all the potential lobbies the clients can choose to join and connect to (see Appendix Listing A.15). Connecting clients retrieve the relay's join code and configure their network transport accordingly to connect to the Relay server's IP address, shown in Appendix Listing A.16.

3.4.5 Network Resilience & Reconnection

Unreliable and intermittent connections between clients and the host will be handled through a multitude of techniques. First, every client's RTT is calculated and displayed to every player with high latency clients being prompted a warning message stating their connection is unstable when in-game. A coroutine runs on every client to regularly send a timestamp to the server to measure their RTT, seen in Appendix Listing A.17. The server then simply echoes back the timestamp to the client. When the client receives the echo from the server, the RTT is

calculated and the UI is updated locally by sending this ping value to the server which broadcasts it to all over players, globally (see Appendix Listing A.18).

Depending on the severity of a client's ping, the solution can automatically reduce the client's application resolution quality and dynamically alter reconciliation and prediction parameters to better suit their connection speed and reliability while also reducing network bandwidth by a lower quality resolution (see Appendix Listing A.20). For example, higher latency players having their reconciliation interval lowered and position error threshold increased, enabling more frequent yet less aggressive corrections in the position and velocity, thereby resulting a smoother experience regardless of network instability.

In addition, giving players who disconnect due to a poor or dropped connection a chance to rejoin the game session, within a specified grace period, and recovering the game state to the last known state the reconnecting client was present for enhances the resilience and state recovery of the networked game. Upon client disconnection, their authentication Id and disconnect time is recorded, as mentioned prior to this section (see Appendix Listing A.19).

The current state of the game is captured and locally saved on the server's device – this state includes player and ball positions, velocities and the game scores. If the disconnected client reconnects within the grace period (of 60 seconds), the server verifies their authentication ID and, if successful, restores them to their previous state, seamlessly allow them to rejoin the ongoing game session (see Appendix Listing A.19).

This approach is suitable as temporary network disruptions do not halt the game entirely, enhancing the multiplayer experience to be fair and resilient. The logic incorporated (grace period and state restoration) is designed to prevent abuse and allow genuine reconnections.

3.4.6 Security & Anti-Cheat

The robust authentication system discussed previously is the foundation for all security and anti-cheat – every player is required to authenticate themselves before any online multiplayer features are accessible. Every action and lobby member is linked to unique PlayerIds and therefore, through the mapping system, every network connection. Only authenticated users can join or create lobbies, perform actions within the network (move, score, ready up) and reconnect to disconnected sessions – ensuring hijacking is not possible.

Server-side validation is constructed on top of this authentication layer; game events are validated by the host/server and are only authorized if they emerge from recognized, authentic clients (see Listing 6). No unauthenticated or spoofed client is able to alter or disrupt the game state. Alongside the server-authoritative logic, this approach accomplishes fair play, accountability of actions and resilience against exploits common in multiplayer games.

```
// Score update method
    if (!IsServer) return; // Only server can update score
    if (!playerIds.ContainsKey(clientId))  {  // Ensure the client is registered and authenticated
        Debug.LogWarning("Score attempt from unregistered client."); // Log warning and return.
    // ... else proceed with scoring logic
```

Listing 6:GameManager Handling Secure Score Update Only for Authenticated and Registered Clients.

3.4.7 Legal, Ethical & Social Considerations

Throughout the entire development and deployment of this project, compliance with data privacy regulations (GDPR), asset licensing and ethical considerations have been a priority.

The solution has been designed to only store and process the absolute minimum necessary data for multiplayer functionality to comply with data protection regulations with respect to player's privacy. Player identities are handles through Unity Authentication which generates anonymous, non-identifiable PlayerIds. Names, emails or other personal details are never used, stored, transmitted or written to disk in this program.

All other information (PlayerIds, session tokens, game states) are only stored temporarily in memory and purged when they have served their purpose – the ReconnectManager records disconnects via anonymous authIds and old data is deleted upon reconnection or the grace period timing out.

Moreover, no IP addresses are shared throughout the networking flow as all connectivity is routed through Unity's Relay service acting as a secure middleman, preventing any direct peer-to-peer connections and protecting users from their personal network information being available. To further reinforce this, the gameplay synchronization is limited to non-sensitive state information and communication is handled by Unity's secure transport layers. This secure design insists on privacy for all players with no unnecessary data collection or retention while fully complying with the legal and ethical standards involved in multiplayer gaming.

Finally, asset usage strictly adheres to licensing terms and conditions. A positive and fair online community is advocated for by limiting the harmful actions players can commit while enforcing a code of conduct and communicating data usage policies to players. Overall, these measures collectively establish a legally complaint, ethically and socially responsible application.

3.5 Summary

A modular, manager-based methodology was deeply discussed and implemented throughout this chapter, with the aim of addressing the challenges of real-time multiplayer networking with physics-based gameplay. A client-

server model that follows server-authoritative logic was leveraged inside the Unity engine, alongside the integration of advanced networking techniques of client prediction, server reconciliation, dynamic parameter adjustment and state recovery.

Unity's tools and services have been incorporated into the project for player identification, seamless matchmaking and reliable connectivity assurance despite suboptimal connection conditions. Moving onwards to Chapter 4, the testing strategy and validation is covered.

Chapter 4: Testing & Validation

4.1 Testing Strategy

Testing was carried out iteratively throughout development where each new implementation, feature or change in architecture was validated by targeted tests. The testing strategy, shown in Table 3, adopted a systematic, multi-stage approach to verify the multiplayer network's reliability, fairness, resilience and functionality under a wide range of conditions. Requirements both functional and non-functional were addressed, including network resilience, security, scalability and synchronization.

Table 3: Test Case Strategy.

Test #	Test Type	Description	Expected Outcome
1	Unit	CoreManager initializing all managers and references.	All managers are instantiated, referenced and accessible upon game start.
2	Integration	Manager references and states persist through scene transitions.	Managers remain as singleton (no duplicates) and their states persist after scene transitions.
3	Functional	Player signing in with a username.	Player authenticates themselves with an inputted username and navigated to menu screen.
4	Functional	Player creates a lobby.	Player starts hosting through NetworkManager and Relay allocation and navigated to lobby screen.
5	Functional	Player joins a lobby by code.	Player starts as a client through NetworkManager and connects to the correct Relay server where they are navigated to lobby screen.
6	Functional	Player searches for lobbies and joins one.	Player sees list of available lobbies and can choose to join one. Player then starts as a client through NetworkManager

			and connects to the correct Relay server and navigated to lobby screen.
7	UI/UX	Lobby search function displays a list of found lobbies.	Upon clicking the search lobbies button, the UI is updated to show a list of available lobbies to join.
8	UI/UX	Lobby screen displays information about current lobby.	Upon navigation to lobby screen, UI is updated to show lobby code, lobby name, player count and a correct list of players in the lobby.
9	Functional	Player readies up in lobby.	Player toggles their ready status via button press and shares status to other players.
10	UI/UX	Player's ready status displayed in player list.	Player's current ready status correctly displayed with their associated player list item.
11	Functional	Host can start the game.	LobbyManager checks whether all players in the lobby are ready – if so all players transition to gameplay scene.
12	Functional	In-game movements and scoring are synchronized across all players.	All clients see same view for all player objects positions, score and events.
13	Network Simulation	Simulate 100, 200ms with 5% packet loss through Network Simulator.	Game remains playable for all clients with no major drift in synchronization.
14	Stress Test	Force a constant heavy load for the network via heavy variable sharing at a fixed rate.	No crashes, network bandwidth and CPU usage within limits.
15	Security	Attempt a spoof of ready status and score update via unauthorized RPCs.	Server rejects all invalid actions and logs warnings.
16	Functional	Client can reconnect after disconnection.	Upon reconnection, player is checked if they are the same player that disconnected – if so their state is restored and the gameplay resumes.
17	UI/UX	Ping values and connection warnings display in UI.	UI for ping display is correctly updated in real-time, warning popups shown.

Unit and integration testing was conducted on every manager and network component in both isolation and in combination with each other to validate correct initialization, handling of references and functionality with scene transitions. Functional testing for the multiplayer system included connecting multiple clients locally and over Relay to assess lobby creation, joining (by code and search) and management alongside ready statuses and player/lobby lists in the UI.

Also, the in-game multiplayer was tested for synchronized gameplay combined with Unity's Network Simulator tool to involve artificial latency, jitter and packet loss to validate the advanced networking capabilities used in the solution (client prediction, reconciliation and dynamic parameter tuning). This allowed for more suitable testing under a wide range of realistic and suboptimal network conditions, validating that every type of condition still ensures synchronization, fairness and stability, except the poorest connection conditions which inevitably suffer a rougher experience and are inherently challenging.

Furthermore, a dedicated network stress test scene was implemented for systematically evaluating bandwidth, latency effects and overall network performance. This was carried out by spawning multiple artificial clients and player objects to measure the usage of network bandwidth, CPU/memory usage and any difference in synchronization whilst the network was under a heavier load.

Likewise, security and anti-cheat checks were handled by attempting to spoof actions; bypassing authentication and manipulating variables. This validated the server-authoritative logic as well as the validation mechanisms (username input, starting game when all players ready). The reconnection system was tested by both manually disconnecting via the menu and by forcibly losing connection, then reconnecting clients to measure the effectiveness of this functionality and the state recovery.

An all-inclusive testing strategy was adopted – stress testing, simulated network conditions and evaluation metrics – were suitably designed for validating whether the solution meets the objectives – synchronization, fairness, resilience and scalability. Security, privacy and ethical considerations have been deliberately thought of during design and development. Next, the results and analysis of these implementations will be presented, with demonstrations of their effectiveness under a range of network conditions.

4.2 Evaluation Metrics

To effectively assess the networking solution's key requirements, a set of qualitative and quantitative metrics were defined and used during the testing and validation process. These metrics were selected to reflect the aims and objectives of the solution while delivering insights into the synchronization, resilience, security and scalability effectiveness. Metrics such as synchronization drift, RTT and packet loss rate will be measured both with and without prediction / reconciliation enabled to gauge how effective these techniques are at reducing overall latency – see Table 4.

Table 4: Evaluation Metrics.

Metric	Purpose	Measurement Method

Synchronization Drift	Interpret the average and maximum difference between the clients' and server's states. Lower drift equates to better synchronization and fairness.	Compare the server's state to each client's state at fixed intervals via RPC's.
Round-trip Time / Ping	Establish connection quality and amount of input delay.	Measure each client's ping via echo requests and responses.
Packet Loss Rate	Discover the percentage of packets lost in transmit – high loss reduces smoothness and reliability.	Simulate packet loss through Unity's Network Simulator and assess the sent/received messages between clients and the server.
Fairness	Measure differences in the game's playability and outcomes due to instability of the network, lag or cheating.	Playtest the game through play sessions with alternate player counts and network conditions via GameManager's scoring.
Bandwidth Usage	Find the amount of data transmitted during typical and stressful activity.	Monitor Unity's Profiler and relevant network logs and measure bandwidth usage directly.
Reconnection Success Rate	Identify the effectiveness of the reconnection system.	Count the successful state restorations and reconnections through ReconnectManager and relevant logs.
Latency Adaptability	Discover how effective the parameter adjustment is on managing synchronization drift and reliability for high-latency clients.	Compare the drift, reliability and user feedback before and after dynamic parameter and resolution adjustment.
Security	Detect the number of detected and blocked unauthorized or spoofed actions.	View server logs and validation checks in GameManager / LobbyManager.

Chapter 5: Results & Validation

5.1 Introduction

This chapter exhibits the outcomes of the testing and validation conducted on the multiplayer networking solution as described in Chapter 4. The evaluation is centred around verifying whether the implementation meets the aims and objectives of the project, specifically synchronization accuracy, network resilience, security and user experience under various network conditions.

Both quantitative and qualitative results are reported here, with each test case and metric for evaluation corresponding to the requirements and objectives outlined in earlier chapters. These results are organized by test type and metric via the summary table (see Table 5) and visualisations for clarity. No discussion on the actual results is included here – only the factual outcomes which will later be analysed in Chapter 6.

5.2 Summary of Test Outcomes

Table 5: Test Case Results.

Test #	Test Type	Description	Actual Outcome	Pass/Fail
1	Unit	CoreManager instantiation, references.	Correct instantiation, references and accessibility.	Pass
2	Integration	Scene transitions preserve manager state and references.	No duplicate managers, states have persisted.	Pass
3	Functional	Player sign-in.	Successful authentication with inputted username and navigation.	Pass
4	Functional	Player creating lobby.	Host through NetworkManager and a Relay allocation, navigation.	Pass
5	Functional	Player joining lobby via code.	Client through NetworkManager and connection to Relay server, navigation.	Pass
6	Functional	Player searching lobbies and joining one.	Visible list of available lobbies. Client through NetworkManager and connection to correct Relay server, navigation.	Pass
7	UI/UX	UI lobby list update.	Correct UI update.	Pass
8	UI/UX	UI lobby information update.	Correct UI update.	Pass
9	Functional	Player toggling ready status.	Correct toggle status and sharing.	Pass
10	UI/UX	UI player ready status update.	Correct UI update.	Pass
11	Functional	Host starts game.	Correct check and scene transition.	Pass
12	Functional	In-game synchronization across all clients.	Clients have the same view.	Pass
13	Network Simulation	Artificial network conditions.	Game does remains playable, no major drift in synchronization.	Pass
14	Stress Test	Network Stress Test	No crashes, network bandwidth and CPU usage within limits.	Pass
15	Security	Spoof attempts.	Invalid actions rejected, warnings logged.	Pass

16	Functional	Reconnection functionality.	Player is checked, state restored and gameplay resumed.	Pass
17	UI/UX	UI ping and warning update.	Correct UI update.	Pass

5.3 Results of Evaluation Metrics

5.3.1 Synchronization Drift

Figure 17 and 18 both capture the effect that applying client prediction and server reconciliation has on the amount of synchronization drift during varied network latency. Figure 17 showcases the average and maximum drift values in a table with conditional formatting to highlight the higher drift values in red and lower values in green. This acts as a visual aid to make it easier to identify what techniques are most significant at reducing the amount of drift.

Figure 18 compliments this table by visualizing the same data as a bar chart – this makes the trend of decreasing drift due to the introduction of the networking techniques more prominent. This combination of table and chart allows for precisely comparing the values and intuitively understanding the improvement.

The network latency starts at around 30ms due to Unity's Relay service, making this value the baseline, with 50ms and 100ms being simulated by Unity's Network Simulator. Results are shown for no prediction/reconciliation, prediction only and then both techniques applied.

Figures 17 and 18 portray that, at baseline latency, average drift is 92ms without prediction, reduced to 61ms with prediction and further reduced to 25ms with both techniques. As simulated latency was introduced, the benefits of the techniques become more pronounced – at 100ms latency the average drift was reduced from 261ms to 72ms.

Technique	30ms Avg	30ms Max	50ms Avg	50ms Max	100ms Avg	100ms Max
No Prediction/Reconciliation	92	161	139	211	261	480
Prediction	61	99	104	175	159	253
Prediction + Reconciliation	25	75	41	92	72	120

Figure 17: Average & Maximum Synchronization Drift Before and After Techniques, Under Different Conditions.

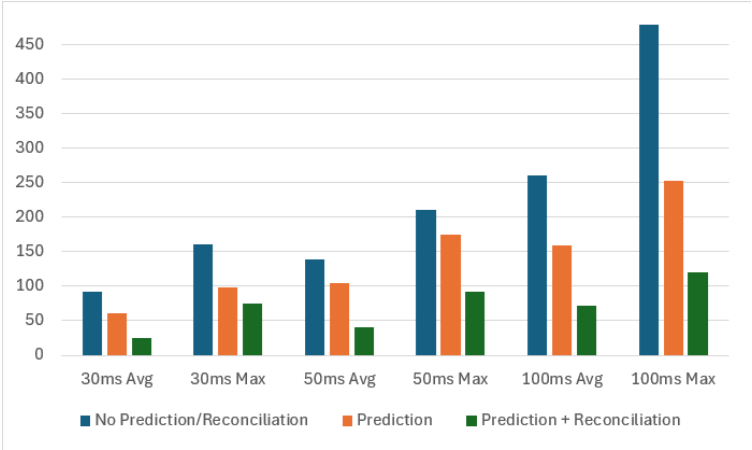

Figure 18: Bar Chart for Change in Synchronization Drift, ms (Y axis) With/Without Techniques, Under Different Conditions.

5.3.2 Client Ping

Figure 19 displays the measured average, minimum and maximum round-trip time for multiplayer networked sessions of varied player counts, tested at the baseline relay latency of 30ms up to simulated latencies of 50ms and 100ms. The line graph in Figure 20 visualizes how these averages, minimums and maximums increase due to higher player counts and increased connection latency. Each line represents a player count and thicker guidelines have been placed at 30ms, 50ms and 100ms to highlight the latency levels used in testing.

Condition	Metric (Ping)	2 Players	4 Players	6 Players	8 Players
Baseline (30ms)	Avg	33	34	35	36
	Min	30	31	32	33
	Max	38	39	40	41
50ms Simulated	Avg	55	57	59	61
	Min	51	52	53	54
	Max	60	62	65	68
100ms Simulated	Avg	105	108	112	115
	Min	100	101	104	106
	Max	112	115	120	125

Figure 19: Average, Minimum and Maximum Ping(ms) for Sessions with 2, 4, 6 and 8 Players Under Latency Conditions

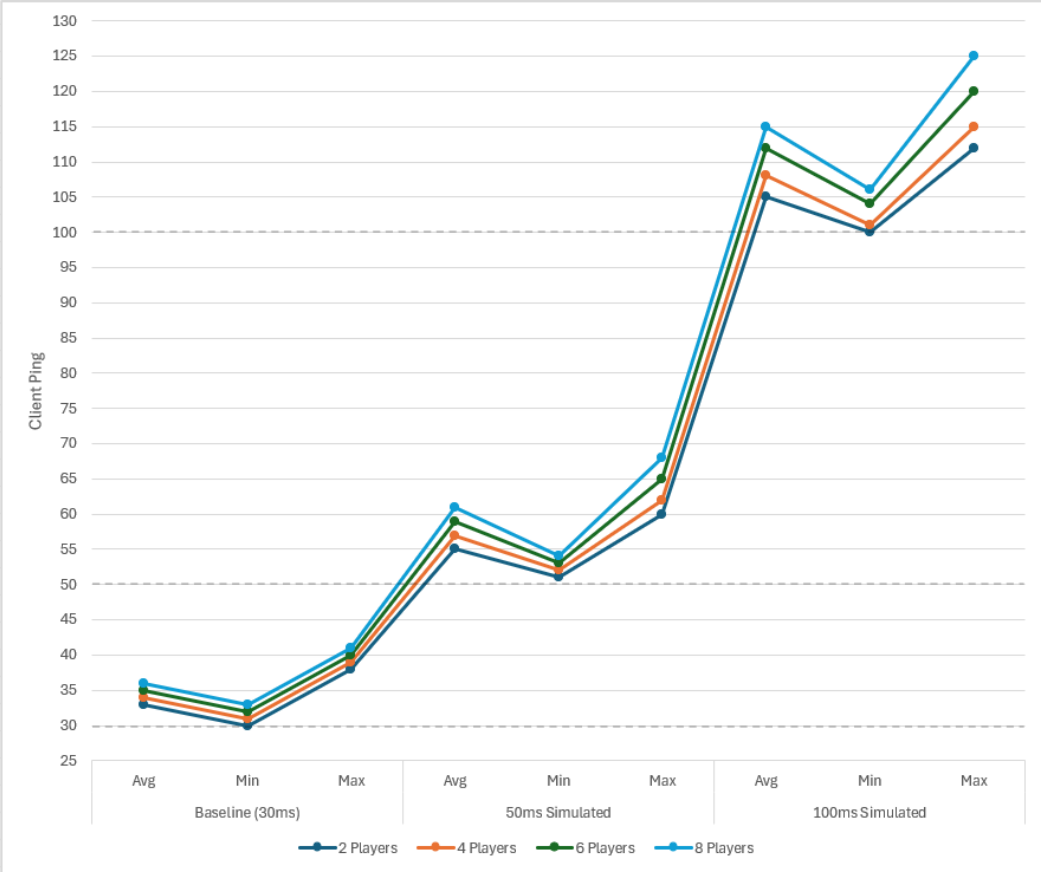

Figure 20: Ping(ms) by Player Count and Various Latency Conditions

Together, Figures 19 and 20 summarize the impact of player count and simulated connection latency on client's ping values in multiplayer playtesting sessions. Conditional formatting in Figure 19 showcases the lower latency connections with full signal bars and in green, with high ping connections having fewer signal bars and in red - showing how ping increases as player counts and connection latency increase.

Figure 20 clearly visualizes these average and peak values and how different they are from the actual latency of the connection with the thicker guidelines helping to relate the measured ping to the intended latency. This combination demonstrates that the network maintains acceptable ping for every network condition tested.

5.3.3 Packet Loss & Fairness

Figures 21 and 22 illustrate the relationship between the amount of packet loss and level of fairness in a multiplayer session. Figure 21 presents the measured fairness metric of average score difference between players across various artificial packet loss rates induced by Unity's Network Simulator. Figure 22 visualizes this trend and shows how an increase in packet loss rate correlates with a reduction in fairness – as packet loss rises, the score difference increases alongside it.

Packet Loss (%)	Score Difference
0	0.2
2	0.5
4	1.2
6	1.9
8	3.1
10	4.2

Figure 21: Packet Loss and the Difference in Score.

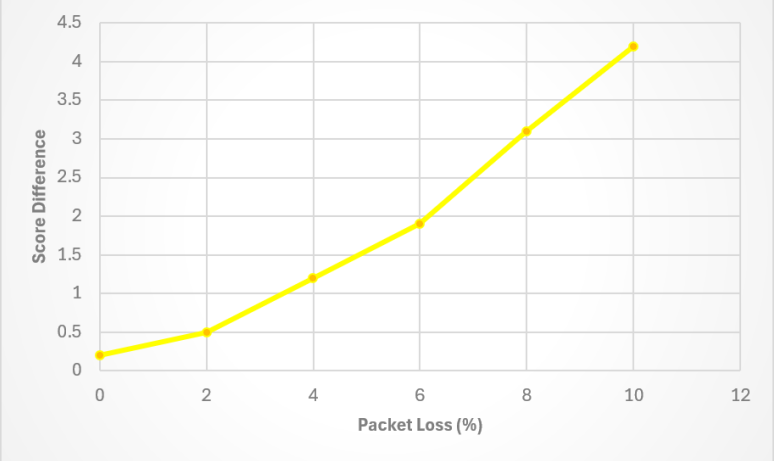

Figure 22: Effect of Packet Loss on the Score Difference.

5.3.4 Bandwidth & Network Adaptability

Figure 23 exemplifies the network's bandwidth usage and latency adaptability under both typical and stressful test scenarios for a varied number of players. The data shows that as the player count is increased, both bandwidth usage and drifts in synchronization increase, increasing much more during the network stress test. This offers a comparison of performance metrics and a view on how much traffic there is within the network, with a look at how the network adapts to these varied amounts of stress.

Condition	Player Count	Avg Bandwidth (MB/s)	Max Bandwidth (MB/s)	Avg Drift (ms)	Avg Ping (ms)
Typical	2	0.8	1	33	33
	4	1.2	1.5	34	34
	6	1.6	2	35	35
	8	2	2.6	36	36
Stress Test	2	1.7	2.4	60	58
	4	2.3	2.7	86	84
	6	2.8	3.6	120	112
	8	4.1	5.1	154	145

Figure 23: Bandwidth Usage & Latency Adaptability under Typical & Stressful Sessions

5.3.5 Reconnection Success Rate & Security

Figure 24 displays the successful reconnection attempts under a variety of network conditions. The data shows that the reconnection attempts remain 100% successful until the artificially induced 100ms latency where one out of 5 attempts fail and worsens during the stress test with the success rate only being 60%.

Condition	Attempts	Successful	Success Rate (%)
Baseline (30ms)	5	5	100
50ms Simulated	5	5	100
100ms Simulated	5	4	80
Stress Test at Baseline	5	3	60

Figure 24: Reconnection Success Rate Under Various Conditions.

Figure 25 summarizes the results of security tests which include attempted spoofing of ready status, unauthorized score changes and reconnecting as another client. Every exploit was successfully blocked due to the server-authoritative logic of the network architecture.

Test #	Test Type	Exploit	Expected Result	Actual Result	Pass/Fail
1	Spoofing	Fake ready status via RPC	Rejected by server	Rejected, warning log	Pass
2	Variable Tampering	Fake score submission	Ignored by server	Ignored, warning log	Pass
3	Auth Bypass	Reconnect as another player	Blocked by server	Blocked	Pass
4	Unauthorized Join	Connect without auth	Denied join	Denied	Pass

Figure 25: Reconnection Success Rate Under Various Conditions

5.4 Summary

This results chapter has unveiled the outcomes of the systematic testing and validation conducted on the multiplayer network, including synchronization drift, client ping, bandwidth usage, success rate of reconnection and security. Quantitative results have been reported for evaluation metrics under various network conditions and player counts. Qualitative results were summarized for reconnection and security tests. The implications of these findings are discussed in the proceeding chapter.

Chapter 6: Discussion & Analysis

6.1 Interpretation of Results

Henceforth, this chapter discusses and analyses the results gathered from the testing and validation carried out by delving into the qualitative and quantitative findings reported in Chapter 5. These results are interpreted in relation to the project's aims and objectives, established requirements and the implications of multiplayer game development in broader terms. A deep understanding of the effectiveness and suitability of the techniques and approaches involved is explored, setting the foundation for the final conclusions.

Based on the results found in Figures 17 and 18, a significant reduction in the amount of synchronization drift is heavily indicated when both client prediction and server reconciliation are applied together.

Furthermore, at 100ms latency that was simulated, this reduction was even evident from 261ms to 72ms. This provides a clear demonstration of the effectiveness of client prediction and reconciliation in minimizing the negative effects that network latency causes, allowing for a more smooth and fair multiplayer experience. This trend aligns with the findings of Lim (Lim Q W 2017) and meets the requirements for real-time multiplayer responsiveness outlined in the project aims.

Figures 19 and 20 give a critical insight into the perceived latency and responsiveness of the game where the average ping remain low and only slightly above the actual latency of the network. Only a minor value of ping on average were client's round-trip times above the baseline network latency, reflecting on the effectiveness of Unity's Relay service and the optimization of the network. When network latency was simulated, the ping values increase alongside it, which is expected. The min and max values at these induced latencies have a wider range compared to the baseline conditions of the network, however the averages remain very close to the actual latency and just slightly more than at the baseline.

The average ping values stayed below the threshold which should generally be avoided for real-time gameplay, even with artificial latency and up to 8 players in the testing session. This confirms that the networking solution delivers on the stable and resilient latency across alternate network conditions – a key requirement for fair play and exceptional user experience in multiplayer games. All players were ensured responsive gameplay and game state updates were not degraded noticeably during playtesting.

Due to Figures 21 and 22, the results highlight that all players experienced equal game states and outcomes, demonstrating the fairness despite induced packet loss. At 0% packet loss, the difference is a minor 0.2 in game score but this is most likely coincidence and not related, therefore this is dismissed.

Only at 6% up to 10% packet loss do the game score differences sharply increase, which is expected as players essentially losing 6-10% of their inputs would lead to this outcome. This reveals that the client prediction and reconciliation aid to alleviate the effects of minor to moderate packet loss, but the higher the rate of packet loss, the greater the divergence in fairness.

It is important to note that generally over 5% packet loss is considered an unstable connection and that specifically real-time, physics-based multiplayer gameplay requires more inputs over a shorter period of time compared to other game genres. Despite this, the networking solution is very effective below 6% packet loss, ensuring that clients with moderately poor connections do still have a fair playing field. The scores, as previously mentioned, likely have some external factor unrelated to the network's efficiency in their outcomes due to playtesting (some play testers have played before and understand the gameplay or are just more experienced at real-time, physics-based gameplay).

Looking at Figure 23, the results divulge that the average bandwidth usage steadily increased with player count. The values are typical for home broadband connections, evidencing that the network is efficient and suitable for multiple players (at least up to a small group due to testing only going up to eight players).

During conditions of the stress test, with highly frequent state updates, the eight-player session bandwidth usage peaked. Even after this higher load and stress, the network maintained performance and functionality as the peak value was only moderate. The average ping and synchronization drift stayed within playable thresholds, ensuring responsive gameplay. The system's adaptive parameters for the networking techniques (prediction, reconciliation) combined with automatic graphical resolution reduction for high latency clients supported this playability under suboptimal conditions. This validates that the solution meets the requirements for network adaptability and scalability needed to meet the aims.

Examining Figure 24, the system displays that the reconnection system builds on the network's resilience over a range of player counts and connection conditions. During the typical and moderate network conditions, the success rate of reconnection was 100% with disconnected clients rejoining and having their prior game state restored accurately within the grace period.

Under stress test conditions with higher latency and packet loss, most attempts were successful as some failures due to connection instability. This proves how effective the ReconnectManager's implementation was in persistent state capture and seamless recovery.

Security testing was conducted via attempts to exploit and cheat the system, as well as spoofing the authentication system. As seen in Figure 25, all exploit and cheat attempts are checked and successfully blocked by the system's server-authoritative design.

All spoofing attempts (unauthorized ready status, score and authentication bypass) were consistently rejected by the server. Direct peer-to-peer connections have been avoided due to Unity's Relay service, aligning with best practices for multiplayer security. Gameplay integrity and player data were protected due to these security mechanisms in the solution.

6.2 Significance of Findings

The outcomes of this solution attest that the carefully considered choices of architecture and networking techniques monumentally elevate the overall performance, quality and resilience of real-time, multiplayer experiences.

Due to the embraced modular, manager-based system that manages all services (networking, authentication, lobbies, ping monitoring and anti-cheat) this system showcases how a practical separation of concerns and central initialization of services streamlines development and maintenance for complex multiplayer games such as this. Additionally, this foundation puts forth feature expansion in the future with easy debugging.

Moreover, the implementation of dynamic adaptation techniques (real-time adjustment of parameters used in prediction, reconciliation and graphic fidelity) spotlights the value of responsiveness and feedback centred on users based on their connection quality.

This system can recover from disconnection, securely authenticating players and sustaining server authority for all crucial gameplay logic which bolsters fairness and security, as well as minimizing the risk of exploitation to achieve a level playing field. Through undeviating enforcement of these principles, the solution demonstrates how a thoughtfully built server-authoritative model can lead to upmost integrity for player data and game state, even with off-the-shelf services.

Above all, the findings further focus attention on the practical feasibility of expanding Unity's networking infrastructure to not only tackle use cases most typical, but also unfavourable conditions such as packet loss, high latency and stressful scenarios without massively compromising the playability or fairness of the experience.

Without a doubt does this work serve as an exhibition that modern Unity games, when assembled with scrutiny to security, modularity and adaptive design, can deliver scalable, robust and fair multiplayer experiences that can meet demands commercial and technical alike.

6.3 Limitations

Although the strengths and effective features of this networking solution, a myriad of limitations should be recognized. To begin, scalability was only evaluated for play sessions up to eight players – network performance and playability at higher player counts remains uncertain. The architecture, which is well-structured and modular, depends on Unity's NGO framework and Relay services. This is convenient; however, it introduces dependencies on third-party service availability, potential regional latency variation and quite limited low-level control of the transport parameters.

Pertaining to the adaptability of the network, whilst dynamic adjustment of parameters proved to be very effective for moderate packet loss and latency, more extreme connection conditions resulted in significant desynchronization and occasional failures in reconnection attempts. This reconnection system is constrained by local state persistence with no support regarding host migration or cross-session recovery.

On security and anti-cheat measures, the solution enforces validation through the server, that said the implementation does not include advanced detection for cheating, distributed denial-of-service (DDoS) attacks or more complex exploits beyond the cases that were tested. Furthermore, as the solution leverages Unity's Authentication and Relay, the security and privacy in relation to player data are subject to the policies of these services.

Conclusively, features for user interface (like adaptive resolution, warning messages, score and timer interface elements) are functional, however more refinement is possible to enhance the user experience and accessibility. The overall areas for future improvements are scalability, further resilience to unstable network conditions, tightened security and user experience.

6.4 Summary

To summarize, this chapter has critically reviewed the performance, effectiveness and limitations of the multiplayer networked game by interpreting the outcomes from the thorough testing of synchronization, latency, fairness, resilience, bandwidth, reconnection and security, followed by a deep analysis of how the system proves it has met the core objectives over a range of typical and realistic connections.

Moreover, this discussion has brought out the architectural strengths while addressing areas where further improvements are required. Altogether, the insights presented here give a comprehensive understanding of the effectiveness of the solution and the foundation it lays for future development, clearing the way for the final conclusions.

Chapter 7: Conclusions & Future Work

7.1 Conclusions

The project aims to design and develop an efficient real-time multiplayer networked game with physics-based gameplay in Unity and sets out to achieve seamless synchronization, fairness and resilience to issues relating to connectivity, as well as tight security. This solution addresses the technical challenges identified at the beginning by combining a modular architecture with Unity's NGO framework and services alongside advanced networking techniques.

Through systematic testing and validation, the solution proves that it can deliver accurate synchronization with low-latency, responsive gameplay for various network conditions and player counts.

Techniques like client prediction and server reconciliation greatly lowered the synchronization drift, with adaptability built into the network to uphold playability and fairness even for higher latency clients with packet loss. The usage of bandwidth scaled as expected with player count and stayed within reasonable thresholds – stress test scenarios also demonstrate that this architecture can manage stressful traffic without critical failure.

The reconnection system further validates the resilience of the network – player states were restored reliably after disconnection. Tests for security confirmed that the client-server topology was competent at validating vital game logic centrally and blocking exploit attempts whilst maintaining data integrity.

Additionally, the solution supported the legal and ethical standards by protecting identities of players and ensuring personal data usage was kept minimal throughout the multiplayer life cycle by Unity's secure services.

Overall, the objectives were met; the system allows for fair, synchronized multiplayer gameplay, supports reconnection and established anti-cheat and security measures. The project also included successful design and implementation of network protocols while acknowledging the difficulties of building a networked game and the mitigation techniques to overcome them.

7.2 Future Work

Elaborating on the achievements as well as the limitations recognized, a multitude of directions for future improvement and research are pronounced. One major element to expand would be the scalability in testing as I only ran play testing sessions for up to a maximum of eight players. This would allow me to be more confident of the system for larger player counts and more advanced game modes, demanding more optimized bandwidth usage, state synchronization and possibly use an alternative or hybrid framework to lower the system's reliance on Unity's assets.

Furthermore, host migration and seamless recovery from repeated disconnections alongside extended gameplay elements would improve resilience and complexity of the networked game while ensuring the gameplay remains playable in less stable conditions. Security could be tightened via advanced anti-cheat techniques like behavioural analysis or real-time monitoring for DDoS attacks.

Refining the network's adaptive features by improving feedback and providing more robust resolution scaling for high ping clients would enhance playability and accessibility. Moreover, an expansion on the modular architecture to allow for cross-platform play, persistent player profiles and adaptive tuning based on analytics would be beneficial. Overall, these future expansions would consider the limitations of the current system and result in a more resilient, scalable and secure foundation.

Chapter 8: Reflection

Firstly, this project has been a personal and technical transformative learning experience. At the conception, my main motivation was to confront the challenges of multiplayer networking within a modern game engine and from the beginning I realized that to succeed would not only require technical proficiency, but also critical

thinking and adaptability due to the complex nature of multiplayer networking, especially real-time, physics-based games.

It was essential of me to decide on multiple key decisions that determined the outcome. One of which was adopting a modular architecture – a pivotal choice as it separated out concerns and allowed for iterative development. During the implementation, I learned that a balance of utilising third-party frameworks (NGO and Relay) along with custom integrations for more advanced networking techniques such as client prediction, server reconciliation, dynamic network resilience and reconnection was crucial. On top of that, I now understand how important thorough research and building on proven practices found in the industry while remaining open-minded to innovation is.

Along many lessons learned, the necessity of extensive testing and validation was one of the most significant. I was able to address the limitations of my initial prototypes by simulating realistic and extreme network conditions to evaluate metrics which proved vital in refining the first designs and prototypes. Additionally, I gathered a deep appreciation for the legal, ethical and social considerations inherent in ensuring fair play and handling user data.

Looking at my aims and the results, I am proud of how the finished system aligns with the outlined objectives despite some features being transformed or reprioritized because of practical constraints. For approaching a similar project again, I would venture further time into the early prototyping stage as this guided the development greatly. In the end, this project has reinforced my technical skills, problem-solving capabilities and confidence in taking on complex, open-ended projects, getting me ready for software and networking engineering, as well as game development in the future.

References

Gambetta, G. (2024). 'Fast-Paced Multiplayer (Part I): Client-Server Game Architecture'. *Gabriel Gambetta*. Available at: https://www.gabrielgambetta.com/client-server-game-architecture.html (Accessed 27 April 2025).

Geeksforgeeks. (2024). 'Differences between TCP and UDP', *Geeksforgeeks*. Available at: https://www.geeksforgeeks.org/differences-between-tcp-and-udp/ (Accessed 27 April 2025).

Geeksforgeeks. (2025). 'Types of Network Topologies'. *Geeksforgeeks*. Available at: https://www.geeksforgeeks.org/types-of-network-topology/ (Accessed 28 April 2025).

Gao, Y. (2018). 'Netcode Concepts Part 1: Introduction'. *Medium*. Available at: https://meseta.medium.com/netcode-concepts-part-1-introduction-ec5763fe458c (Accessed 28 April 2025).

Kurose, J.F. & Ross, K.W. (2017). '*Computer Networking: A Top-Down Approach*' (7th ed). *Pearson*.

Lim, Q.W. (2017). 'How Do Multiplayer Games Sync Their State? Part 1'. *Medium*. Available at: https://medium.com/@qingweilim/how-do-multiplayer-games-sync-their-state-part-1-ab72d6a54043 (Accessed 28 April 2025).

Massive Realm. (2024). 'The Foundation of Realtime Multiplayer, Part 2: Data Transmission Protocols'. *Medium*. Available at: https://medium.com/@massiverealm/the-foundation-of-realtime-multiplayer-part-2-data-transmission-protocols-33eb8e91242b (Accessed 27 April 2025).

Mirror Networking. (2024). 'Mirror Networking'. *GitBook*. Available at https://mirror-networking.gitbook.io/docs (Accessed 29 April 2025).

Motion Picture Association. (2021). 'THEME Report'. *MPA*. Available at: https://www.motionpictures.org/wp-content/uploads/2022/03/MPA-2021-THEME-Report-FINAL.pdf (Accessed 2 May 2025).

MY.GAMES. (2023). 'Unity Realtime Multiplayer, Part 1: Networking Basics'. Medium. Available at: https://medium.com/my-games-company/unity-realtime-multiplayer-part-1-networking-basics-88226b961a95 (Accessed 28 April 2025).

Photon. (2025). 'Introduction'. *Photon Engine*. Available at https://doc.photonengine.com/pun/current/getting-started/pun-intro (Accessed 29 April 2025).

PwC. (2022). 'Perspectives from the Global Entertainment & Media Outlook 2022-2026'. *PwC*. Available at: https://www.pwc.com/gx/en/industries/entertainment-media/outlook/downloads/PwC_Outlook22.pdf (Accessed 2 May 2025).

Ritchie, H., Mathieu, E., Roser, M. and Ortiz-Ospina, E. (2023). 'Internet', *Our World in Data*. Available at: https://ourworldindata.org/internet (Accessed 27 April 2025).

Siddiqui, L. (2024). 'What Is Network Architecture?'. *Splunk*. Available at: https://www.splunk.com/en_us/blog/learn/network-architecture.html (Accessed 27 April 2025).

Sparrow, A. L., Gibbs, M. & Arnold, M. (2020). 'Ludic Ethics: The Ethical Negotiations of Players in Online Multiplayer Games'. *Sage Journals*. Available at https://journals.sagepub.com/eprint/QZW6DGXJBBYZMFXJ9ACY/full (Accessed 29 April 2025).

Stagner, A.R. (2013). *'Unity Multiplayer Games'*. *Packt Publishing*.

summertimeWintertime. (2022). 'Multiplayer Networking Solutions'. *Reddit*. Available at: https://www.reddit.com/r/gamedev/comments/xwnyga/multiplayer_networking_solutions/ (Accessed 28 April 2025).

Tikhomirov, A. (2023). 'Developing an Online Multiplayer Game in Unity'. *South-Eastern Finland University of Applied Sciences*. Available at: https://www.theseus.fi/bitstream/handle/10024/801273/Tikhomirov_Alexander.pdf (Accessed 2 May 2025).

Unity Technologies. (2025a). 'Relay Servers'. *Unity Technologies*. Available at: https://docs.unity.com/ugs/en-us/manual/relay/manual/relay-servers (Accessed 28 April 2025).

Unity Technologies. (2025b). 'About Netcode for GameObjects'. *Unity Technologies*. Available at: https://docs-multiplayer.unity3d.com/netcode/current/about/ (Accessed 29 April 2025).

Unity Technologies. (2025c). 'Unity Authentication'. *Unity Technologies*. Available at:

https://docs.unity.com/ugs/manual/authentication/manual/overview (Accessed 29 April 2025).

Unity Technologies. (2025d). 'Unity Lobby'. *Unity Technologies*. Available at:

https://docs.unity.com/ugs/manual/lobby/manual/unity-lobby-service (Accessed 29 April 2025).

Unity Technologies. (2025e). 'Network Simulator'. *Unity Technologies*. Available at: https://docs-multiplayer.unity3d.com/tools/current/tools-network-simulator/ (Accessed 3 May 2025).

Unity Technologies. (2025f). 'RPC vs NetworkVariable'. *Unity Technologies*. Available at: https://docs-multiplayer.unity3d.com/netcode/current/learn/rpcvnetvar/ (Accessed 5 May 2025).

Valve Corporation. (2024). 'Networking in Source Engine Multiplayer Games'. *Valve Software*. Available at: https://developer.valvesoftware.com/wiki/Source_Multiplayer_Networking (Accessed 27 April 2025).

Xu, Y., Nawaz, S. & Mak, R.H. (2014). 'A Comparison of Architecture in Massive Multiplayer Online Games'. *Eindhoven University of Technology*. Available at: https://www.researchgate.net/publication/271490933_A_Comparison_of_Architectures_in_MassiveMultiplayer_Online_Games (Accessed 29 April 2025).

Appendix

Note: this appendix includes the project repository, extra figures. Also, all Listings of code mentioned in Chapter 3 to provide a deep coverage of the raw C# code developed for the implementation of the system. Minor omissions are present for clarity.

Full Source Code Repository Available at: https://csgitlab.reading.ac.uk/pg011732/cs3ip

Basis	Transmission Control Protocol (TCP)	User Datagram Protocol (UDP)
Type of Service	TCP is a connection-oriented protocol. Connection orientation means that the communicating devices should establish a connection before transmitting data and should close the connection after transmitting the data.	UDP is the Datagram-oriented protocol. This is because there is no overhead for opening a connection, maintaining a connection, or terminating a connection. UDP is efficient for broadcast and multicast types of network transmission.
Reliability	TCP is reliable as it guarantees the delivery of data to the destination router.	The delivery of data to the destination cannot be guaranteed in UDP.
Error checking mechanism	TCP provides extensive error-checking mechanisms. It is because it provides flow control and acknowledgment of data.	UDP has only the basic error-checking mechanism using checksums.
Acknowledgment	An acknowledgment segment is present.	No acknowledgment segment.
Sequence	Sequencing of data is a feature of Transmission Control Protocol (TCP). this means that packets arrive in order at the receiver.	There is no sequencing of data in UDP. If the order is required, it has to be managed by the application layer.
Speed	TCP is comparatively slower than UDP.	UDP is faster, simpler, and more efficient than TCP.
Retransmission	Retransmission of lost packets is possible in TCP, but not in UDP.	There is no retransmission of lost packets in the User Datagram Protocol (UDP).
Header Length	TCP has a (20-60) bytes variable length header.	UDP has an 8 bytes fixed-length header.
Weight	TCP is heavy-weight.	UDP is lightweight.
Handshaking Techniques	Uses handshakes such as SYN, ACK, SYN-ACK	It's a connectionless protocol i.e. No handshake
Broadcasting	TCP doesn't support Broadcasting.	UDP supports Broadcasting.
Protocols	TCP is used by HTTP, HTTPs , FTP , SMTP and Telnet .	UDP is used by DNS , DHCP , TFTP, SNMP , RIP , and VoIP .
Stream Type	The TCP connection is a byte stream.	UDP connection is a message stream.
Overhead	Low but higher than UDP.	Very low.
Applications	This protocol is primarily utilized in situations when a safe and trustworthy communication procedure is necessary, such as in email, on the web surfing, and in military services.	This protocol is used in situations where quick communication is necessary but where dependability is not a concern, such as VoIP, game streaming, video, and music streaming, etc.

Figure A.1: Comparison Between TCP and UDP (Geeksforgeeks 2024).

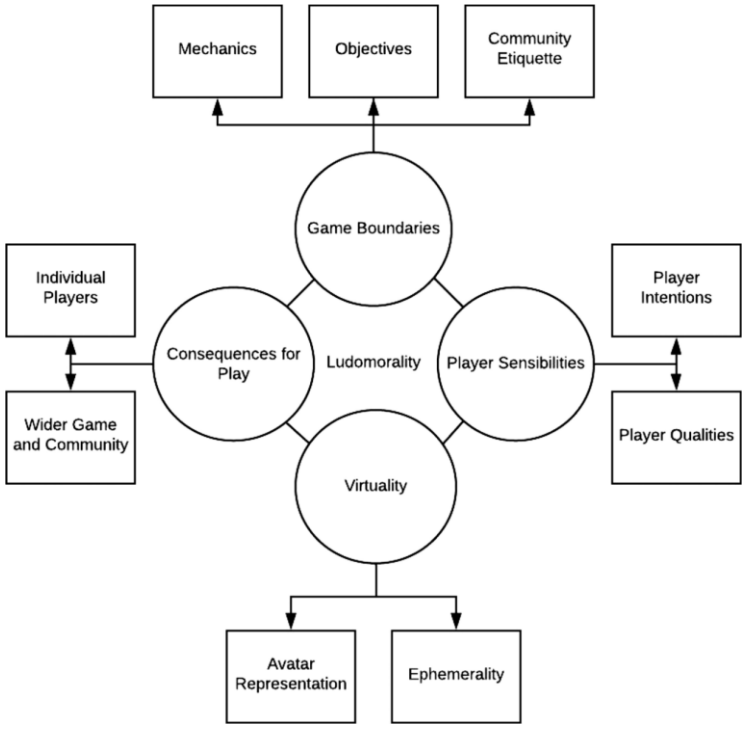

Figure A.2: Four Themes of Ludomorality with Their Subthemes (Sparrow et al 2020).

```
public bool InitializeNetworkManager() {

if (isNetworkInitialized) return true; // If already initialized, exit

if (NetworkManager.Singleton != null) { // Check if NetworkManager.Singleton already exists

networkManagerInstance = NetworkManager.Singleton; // set the reference as that singleton

DontDestroyOnLoad(networkManagerInstance.gameObject); // make it persistent between scenes

isNetworkInitialized = true;

return true; }

if (networkManagerPrefab != null) // If no existing instance, instantiate from prefab {

GameObject networkManagerObject = Instantiate(networkManagerPrefab);

networkManagerInstance = networkManagerObject.GetComponent<NetworkManager>(); // get the
NetworkManager component

// ... same logic as before – make persistent and return
```

```
} Debug.LogError("NetworkManager prefab not assigned in inspector!"); // else output error message
return false; }
```

Listing A.1: CoreManager Securing a Single NetworkManager Instance that is Persistent.

```
private NetworkVariable<int> playerScore1 = new NetworkVariable<int>(0,  // Initial value
  NetworkVariableReadPermission.Everyone,  // Everyone can read the score
  NetworkVariableWritePermission.Server );  // Only server can modify

public void Score(bool isPlayer1Scored)  {
  if (!IsServer) return; // Only server can update score
  if (isPlayer1Scored) // if player 1 scored  {
    playerScore1.Value += 1; // update player 1 score
    if (playerScore1.Value >= WIN_SCORE) { // Check for win condition
      GameOver(1);
      return;  } }
  else  {// else player 2 scored
} // ... same logic as before, GameOver(2) instead
StartResetCountdown();   }  // Reset and start countdown for next point
```

Listing A.2: GameManager - Player Score NetworkVariable Instantiation and Server-Authoritative Score Update.

```
public struct MyCustomData : INetworkSerializable { public int _int; public bool _bool; public string message;
  public void NetworkSerialize<T>(BufferSerializer<T> serializer) where T : IReaderWriter  {
    serializer.SerializeValue(ref _int);
```

```
    serializer.SerializeValue(ref _bool);
    serializer.SerializeValue(ref message);   } }
```

Listing A.3: PlayerNetwork - Custom Data Serialization for Efficient Network Transmission.

```
[ServerRpc] private void RequestAccelerateServerRpc(Vector3 InputKey, ServerRpcParams rpcParams = default) {
  if (!movementEnabled) return; // skip if movement disabled
  Vector3 force = InputKey * acceleration;    // server is authoritative
  rb.AddForce(force, ForceMode.Force);
  if (rb.linearVelocity.magnitude > maxSpeed) {
    rb.linearVelocity = rb.linearVelocity.normalized * maxSpeed;    }
  // Only send updates if position or velocity changed significantly
  if (Vector3.Distance(transform.position, lastSentPosition) > POSITION_THRESHOLD ||
    Vector3.Distance(rb.linearVelocity, lastSentVelocity) > VELOCITY_THRESHOLD)  {
    lastSentPosition = transform.position;
    lastSentVelocity = rb.linearVelocity;    }
  // Create a list to store client IDs except the owner
  List<ulong> targetClients = new List<ulong>();
  foreach (ulong id in NetworkManager.ConnectedClientsIds)  {
    if (id != OwnerClientId) {
      targetClients.Add(id);  } }

var clientRpcParams = new ClientRpcParams  {      // Send to all clients EXCEPT the owner using TargetClientIds
    Send = new ClientRpcSendParams {
```

```
        TargetClientIds = targetClients.ToArray() } };
    SyncMovementClientRpc(transform.position, rb.linearVelocity, clientRpcParams); }
```

Listing A.4: PlayerNetwork - ServerRPC Clients Call to Request Movement of their Player Object from the Server.

```
public async Task Authenticate(string playerName) {
  try  {
    bool isInitialized = await CoreManager.Instance.InitializeUnityServices(playerName);
    if (!isInitialized) {
      Debug.LogError("Authentication aborted: Unity Services failed to initialize.");
      return; }
    AuthenticationService.Instance.SignedIn +=  () =>  {
      Debug.Log($"Signed in as {AuthenticationService.Instance.PlayerId}"); };
    await AuthenticationService.Instance.SignInAnonymouslyAsync();
    Debug.Log($"Authentication complete for player: {playerName}");  }
  catch (System.Exception e)  {
    Debug.LogError($"Authentication failed: {e.Message}");
    throw; } }
```

Listing A.5: MenuManager Handling User Authentication via Unity's Authentication Service.

```
[ClientRpc] private void SyncMovementClientRpc(Vector3 newPos, Vector3 newVelocity, ClientRpcParams
rpcParams = default)  {
    if (IsOwner) return;
    // Update pos for client
```

```
            transform.position = newPos;
            rb.linearVelocity = newVelocity;
            lastReceivedPosition = transform.position;       // Save start and target positions for interpolation
            targetPosition = newPos;
            interpolationTime = 0f;
            rb.linearVelocity = newVelocity; } // Still update velocity immediately
```

Listing A.6: PlayerNetwork – ClientRPC Server Calls After Receiving the ServerRPC to Sync Back Correct Positions and Velocity to Clients.

```
        [ServerRpc(RequireOwnership = false)] public void UpdateReadyStatusServerRpc(string playerId, bool isReady) {
            playerReadyStatus[playerId] = isReady;
            UpdateReadyStatusClientRpc(playerId, isReady); }
```

Listing A.7: LobbyManager Tending to Ready Status Changes for Players in Lobby.

```
[ServerRpc(RequireOwnership = false)] public void StartGameServerRpc() {
            if (!NetworkManager.Singleton.IsHost) return;
            NetworkManager.Singleton.SceneManager.LoadScene("BallArena", LoadSceneMode.Single); }
```

Listing A.8: MenuManager Transitioning All Clients into the Game.

```
        // In Update() method
        if (IsOwner) {
            reconciliationTimer += Time.deltaTime;
            if (reconciliationTimer >= RECONCILIATION_INTERVAL) {
                reconciliationTimer = 0f;
```

```
    RequestServerPositionServerRpc();  } }

[ServerRpc] private void RequestServerPositionServerRpc(ServerRpcParams rpcParams = default) {
  // Server sends its authoritative position/velocity to the requesting client
  ReconcilePositionClientRpc(transform.position, rb.linearVelocity,
    new ClientRpcParams {
      Send = new ClientRpcSendParams= {
        TargetClientIds = new ulong[] { OwnerClientId } } }); }
```

Listing A.8: PlayerNetwork – Reconciliation System Timer and ServerRPC that Clients Call to Request the Position from the Server.

```
[ClientRpc] private void ReconcilePositionClientRpc(Vector3 serverPos, Vector3 serverVel, ClientRpcParams
rpcParams = default) {
  if (!IsOwner) return;    // Only the owner should process this
  Vector3 posDifference = serverPos - transform.position;     // Calculate the difference between predicted and
actual position
  // Only correct if the error exceeds the threshold AND
  // we're not in the middle of applying a previous correction
  if (posDifference.magnitude > positionErrorThreshold && !isCorrectingPosition)  {
    isCorrectingPosition = true;
    StartCoroutine(SmoothCorrection(serverPos, serverVel)); } }
```

Listing A.9: PlayerNetwork - ClientRPC for Owners (Clients) of Player Objects Reconciliation.

```
private IEnumerator SmoothCorrection(Vector3 targetPos, Vector3 targetVel) {
  Vector3 startPos = transform.position;
  Vector3 startVel = rb.linearVelocity;
  float elapsed = 0f;
  float duration = 0.1f; // Correction time in seconds
  while (elapsed < duration)  {
```

```
        elapsed += Time.deltaTime;
        float t = elapsed / duration;
        // Lerp position and velocity
        transform.position = Vector3.Lerp(startPos, targetPos, t);
        rb.linearVelocity = Vector3.Lerp(startVel, targetVel, t);
        yield return null; }
    // Ensure we end exactly at target values
    transform.position = targetPos;
    rb.linearVelocity = targetVel;
    isCorrectingPosition = false;   // Set flag when done }
```

Listing A.10: PlayerNetwork – Smooth Correction Enumerator for Player Positions and Velocity.

```
[ClientRpc] private void SyncMovementClientRpc(Vector3 newPos, Vector3 newVelocity, ClientRpcParams
rpcParams = default) {
    if (IsOwner) return;
    // Save start and target positions for interpolation
    lastReceivedPosition = transform.position;
    targetPosition = newPos;
    interpolationTime = 0f;
    rb.linearVelocity = newVelocity;    // Still update velocity immediately }


// In Update()
if (!IsOwner) {
    interpolationTime += Time.deltaTime * 20f; // Interpolation speed factor
    transform.position = Vector3.Lerp(lastReceivedPosition, targetPosition, interpolationTime); }
```

Listing A.11: PlayerNetwork – ClientRPC For Smooth Movement on Other Clients' Characters.

```
private Dictionary<ulong, string> playerIds = new Dictionary<ulong, string>();
```

```
[ServerRpc(RequireOwnership = false)] public void RegisterPlayerAuthIdServerRpc(string authId,
ServerRpcParams rpcParams = default) {
  ulong clientId = rpcParams.Receive.SenderClientId;
  if (playerIds.ContainsKey(clientId))  {
    playerIds[clientId] = authId; }
  else  {
    playerIds.Add(clientId, authId);}
  Debug.Log($"Registered client {clientId} with authId {authId}");}
```

Listing A.12: GameManager Registering ClientId and PlayerId Mappings for Players Through a ServerRPC.

```
  [ServerRpc(RequireOwnership = false)] public void UpdateReadyStatusServerRpc(string playerId, bool isReady)
  { if (!NetworkManager.Singleton.IsServer) // only executed on the server {
      Debug.LogError("[ServerRpc] UpdateReadyStatusServerRpc called on a non-server instance!");
      return; }
    // Update the server's authoritative ready status
    if (playerReadyStatus.ContainsKey(playerId)) {
      playerReadyStatus[playerId] = isReady; }
    else {
      playerReadyStatus.Add(playerId, isReady);}
    Debug.Log($"[Server] Updated ready status for player {playerId}: {isReady}");
    UpdateReadyStatusClientRpc(playerId, isReady); // Notify all clients about the updated ready status}

  [ClientRpc] private void UpdateReadyStatusClientRpc(string playerId, bool isReady)
  {  // Instead of checking just IsServer, check that this instance is not a pure server
```

```
if (NetworkManager.Singleton.IsServer && !NetworkManager.Singleton.IsClient) {
    Debug.Log("[ClientRpc] UpdateReadyStatusClientRpc called on server-only instance. Ignoring...");
    return; }
// ... same logic as before - update the ready status on all clients,
menuUIManagerInstance.UpdatePlayerReadyStatus(playerId, isReady); }// and update UI
```

Listing A.13: LobbyManager handling ready status toggling and transmission via Server and ClientRPCs.

```
public async Task<bool> JoinLobbyByCode(string lobbyCode) {
    try { JoinLobbyByCodeOptions joinLobbyByCodeOptions = new JoinLobbyByCodeOptions // Join the lobby
using the provided code {
Player = GetPlayer() };
Lobby lobby = await LobbyService.Instance.JoinLobbyByCodeAsync(lobbyCode, joinLobbyByCodeOptions);
joinedLobby = lobby;
        Debug.Log("Joined lobby with code " + lobbyCode);          // Output that the lobby was joined
        PrintPlayers(joinedLobby); // Output player count
        PlayerPrefs.SetString("LastJoinedLobby", lobby.Id);          // Save this lobby as the last joined lobby
        PlayerPrefs.Save();
        bool relayConfigured = await ConfigureRelayForClient(joinedLobby.Data);          // Use the helper method to
configure Relay for the client
        // Start the client
        if (NetworkManager.Singleton != null && !NetworkManager.Singleton.IsListening)
        // After successfully joining a lobby
        OnJoinedLobby?.Invoke(this, new LobbyEventArgs {
            lobby = joinedLobby,
```

```
            PlayerId = AuthenticationService.Instance.PlayerId,
            PlayerName = playerName });
        return true; }
```

Listing A.14: LobbyManager Allowing Players to Join by Lobby Code.

```
  public async Task<List<Lobby>> SearchAndRefreshLobbies() {
   try { QueryLobbiesOptions queryOptions = new QueryLobbiesOptions  {
// set filters – must have open slot, 25 max lobbies  },
          Order = new List<QueryOrder>{
           // set order – ascending, newest first }  };
       QueryResponse queryResponse = await LobbyService.Instance.QueryLobbiesAsync(queryOptions);
OnLobbyListChanged?.Invoke(this, new OnLobbyListChangedEventArgs { lobbyList = queryResponse.Results
});          // Notify UI return queryResponse.Results;
```

Listing A.15: LobbyManager Searching for Lobbies and Updating UI for User Feedback.

```
   private async Task<bool> ConfigureRelayForClient(Dictionary<string, DataObject> lobbyData)
  {
     if (lobbyData.TryGetValue("RelayJoinCode", out DataObject relayJoinCodeData)) // get relay join code
     {
        JoinAllocation joinAllocation = await RelayService.Instance.JoinAllocationAsync(joinCode);
        var transport = NetworkManager.Singleton.GetComponent<UnityTransport>();
        if (transport != null) {
           var relayServerData = AllocationUtils.ToRelayServerData(joinAllocation, "dtls");
```

```
            transport.SetRelayServerData(relayServerData);

            Debug.Log("Configured transport with relay data, connecting as client...");

            return true; }
```

Listing A.16: LobbyManager's Helper Method to Configure Transport According to the Relay Server Found by Join Code.

```
[ServerRpc(RequireOwnership = false)] public void RequestPingEchoServerRpc(float sendTime, string playerId,
ServerRpcParams rpcParams = default) {
   RespondPingEchoClientRpc(sendTime, playerId); // Just echo back the sendTime to the client }
```

Listing A.17: PingManager – ServerRPC to Echo the Ping Request to Clients.

```
[ClientRpc] public void RespondPingEchoClientRpc(float sendTime, string playerId)
{  if (playerId != AuthenticationService.Instance.PlayerId) // Only the client who sent the ping should process this
     return;
   float ping = (Time.realtimeSinceStartup - sendTime) * 1000f;
   playerPing[playerId] = ping;
 ReportPingToServerRpc(playerId, ping); // Send the measured ping to the server so it can update its dictionary
and broadcast to all clients
   UpdatePingUI(playerId, ping);  } // Update local U
```

Listing A.18: PingManager Utilizing a ClientRPC to Calculate the RTT of the Echo and Update UI.

```
// Called by GameManager when a disconnect is detected
public GameState CaptureGameState()
{  GameState state = new GameState(); state.gameTime = Time.time;
   // ... capture additional state (players, ball, scores, etc.)
```

```
  Debug.Log("Capturing game state for reconnection.");

  return state; }
```

Listing A.19: ReconnectManager Capturing Game State.

```
public void changeResolution(float ping)  {  if (Mathf.Abs(ping - lastResolutionPing) < 10) return; // Avoid small fluctuations

    lastResolutionPing = ping;

    Vector2Int currentResolution = new Vector2Int(Screen.width, Screen.height);

    Vector2Int targetResolution;

    // Determine target resolution based on ping and set resolution

    if (ping > 100) {

      targetResolution = new Vector2Int(640, 360);

    } // repeat for > 50, <50

    if (currentResolution != targetResolution) { // Only change if needed

      Screen.SetResolution(targetResolution.x, targetResolution.y, FullScreenMode.FullScreenWindow);  }  }
```

Listing A.20: GameUIManager changing resolution based on ping.